Case: 14-1351 Document: 21 Page: 1 Filed: 05/13/2014

No. 14-1351

United States Court of Appeals for the Federal Circuit

ORACLE AMERICA, INC.,

Appellant,

v.

GOOGLE, INC.,

Appellee.

Appeal from the United States Patent and Trademark Office, Patent Trial and Appeal Board, Reexamination Control No. 95/001,548.

CORRECTED BRIEF FOR APPELLANT ORACLE AMERICA, INC.

MEHRAN ARJOMAND
MORRISON & FOERSTER LLP
707 Wilshire Boulevard
Los Angeles, CA 90017
Telephone: (213) 892-5630
Facsimile: (213) 892-5454

Counsel for Appellant Oracle America, Inc.

May 13, 2014

CERTIFICATE OF INTEREST

Counsel for the Appellant Oracle America, Inc. certifies the following:

1. The full name of every party or amicus represented by me is:

   Oracle America, Inc.

2. The name of the real party in interest (if the party named in the caption is not the real party in interest) represented by me is:

   N/A

3. All parent corporations and any publicly held companies that own 10% or more of the stock of the party or amicus curiae represented by me are:

   Oracle Corporation

4. The names of all law firms and the partners or associates that appeared for the party or amicus now represented by me in the trial court or are expected to appear in this court are:

   Morrison & Foerster LLP: Christopher B. Eide, Mehran Arjomand

Dated: May 13, 2014 /s/ Mehran Arjomand

TABLE OF CONTENTS

TABLE OF AUTHORITIES

Page(s)

CASES

STATUTES

STATEMENT OF RELATED CASES

This action has not previously been before this or any other appellate court. Counsel for appellant Oracle America, Inc. ("Oracle") is aware of the following related case: *Oracle America, Inc. v. Google, Inc.*, Nos. 13-1021, -1022 (Fed. Cir. May 9, 2014) (O'Malley, Plager, Taranto, J.J.).

Oracle filed suit against Google, Inc. ("Google") alleging that Google's Android mobile operating system infringed Oracle's copyrights and patents, including U.S. Patent No. 6,910,205 (the "'205 Patent"), the validity of which is at issue on this appeal. The related case relates to the appeal of the copyright claims; the disposition of the patent claims was not appealed.

Counsel is unaware of any other pending case that will affect or be affected directly by this Court's decision.

JURISDICTIONAL STATEMENT

(a) The statutory basis for jurisdiction of the Patent Trial and Appeal Board (the “Board”) from an appeal by a patent owner in an *inter partes* reexamination proceeding is 35 U.S.C. § 134(b).

(b) The statutory basis for jurisdiction of this Court to hear this appeal is 28 U.S.C. § 1295(a)(4)(A) (as amended by Leahy-Smith America Invents Act, Pub. L. No. 112-29, § 7(c)(2), 125 Stat. 284, 314 (effective Sept. 16, 2011)).

(c) The appeal for review in this case is timely since it was filed within the time provided in 35 U.S.C. § 142 from the final decision of the Board dated November 27, 2013.

STATEMENT OF THE ISSUES

1. Whether the Board erred in construing "overwriting" in claims 2-4, 15, 16, and 18-21 as an act of replacing some information in a computer file with new information, rather than literally writing over an existing information.

2. Whether the Board erred in affirming the Examiner's rejection of claims 2-4, 15, 16, and 18-21 as anticipated by Magnusson when Magnusson does not expressly or inherently disclose any "overwriting" as properly construed.

3. Whether the Board erred in affirming the Examiner's finding that Magnusson is an enabling reference so as to anticipate claims 1-4, 8, 15, 16, and 18-21.

INTRODUCTION

In affirming a sole anticipation rejection, the Board erroneously construed the claim term "overwriting" *not* to mean writing over. This was one of many errors in a cursory decision.

The present appeal arises from an *inter partes* reexamination of the '205 Patent. The '205 Patent is directed to a method for increasing the execution speed of computer interpreters that interpret computer instructions, such as Java bytecodes. The method overwrites an instruction (*e.g.*, a Java bytecode) with a new instruction. The new instruction, in turn, references "native" instructions, which can be more quickly executed.

Google initiated the reexamination by proposing 16 grounds of rejection for original claims 1-4 and 8. The Examiner, however, only adopted an anticipation rejection for claims 1-4 and 8 based on an article entitled "Partial Translation" by Peter Magnusson ("Magnusson") and an anticipation rejection for claim 8 based on an article entitled "The Power of Partial Translation: An Experiment with the C-Ification of Binary Prolog," by Paul Tarau et al. ("Tarau"). In response, Oracle explained that Magnusson was not enabling for claims 1-4 and 8 and failed to disclose "overwriting" an instruction as recited in claims 2-4. The rejection for claim 8 with respect to Tarau was overcome by an amendment to include a feature common with claim 1 and lacking in Tarau. Oracle further added new claims 15, 16, and 18-21, which also recited "overwriting" an instruction. After the Examiner maintained the rejection based on Magnusson in a Right of Appeal Notice dated November 21, 2012, Oracle appealed to the Board.

In its Decision dated November 27, 2013, the Board first addressed the claims that recited the term "overwriting," *i.e.*, claims 2-4, 15, 16, and 18-21. The Board entered an explicit claim construction of "overwriting" that was not previously proffered in *any* paper during the reexamination by Oracle, Google, or the Examiner. Importantly, the Board ignored the plain language of "overwriting," which literally means writing over, and construed the term *not* to mean writing over. Moreover, the Board misread the specification, which—consistent with

Case: 14-1351 Document: 20 Page: 11 Filed: 05/13/2014

"overwriting" meaning writing over—shows an instruction ("GO_NATIVE N#") *writing over* an instruction ("BYTECODE 2") in annotated Figure 5 below.

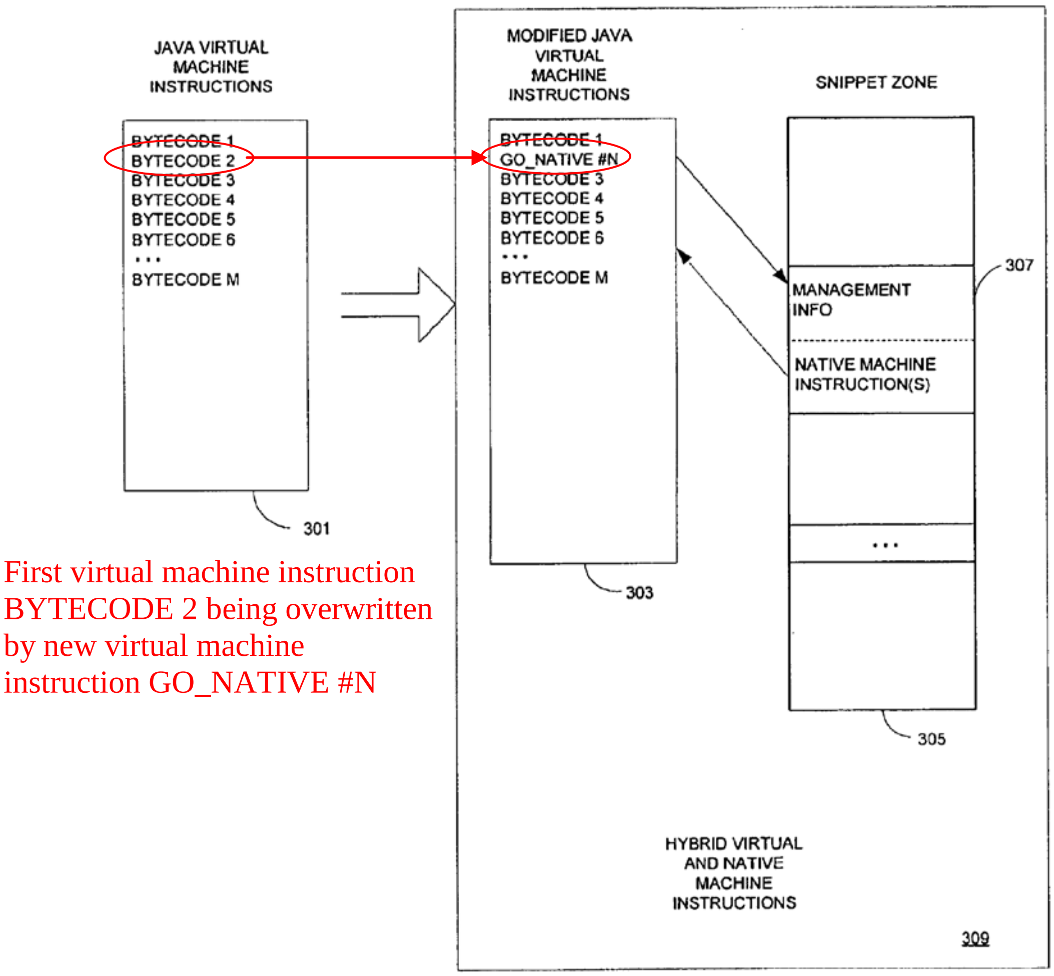


FIG. 5

Under its new claim construction, the Board then affirmed—in two sentences—the Examiner's anticipation rejection of "overwriting" claims 2-4, 15, 16, and 18-21. But under the proper construction, Magnusson does not meet the strict requirements of anticipation. Google throughout the reexamination relied

primarily on the "interpreted program" of Magnusson's Figure 4 to show "overwriting" via an instruction called TRANSLATED. However, as seen below, the rectangular box of the "interpreted program" (shown with a block arrow) does not show any instruction being overwritten by a TRANSLATED instruction and, in fact, does not even show the TRANSLATED instruction. Given this deficiency, Google at times appeared to rely on inherent anticipation, but it has failed to establish that Magnusson *necessarily* requires overwriting by a TRANSLATED instruction.

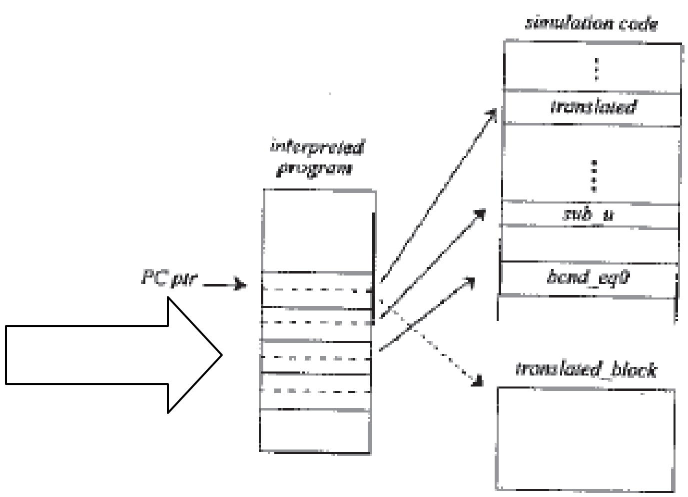


Figure 4 - Partial translation

Finally, the Board found in a cursory fashion that Magnusson was an enabling reference. The Board failed to recognize that Magnusson does not provide an enabling disclosure to a key component of the anticipation rejection, *i.e.*, how Magnusson's TRANSLATED instruction is to be introduced. Based on

Case: 14-1351 Document: 21 Page: 13 Filed: 05/13/2014

this error as to enablement, the Board affirmed the anticipation rejection of not only "overwriting" claims 2-4, 15, 16, and 18-21, but also claims 1 and 8.

STATEMENT OF THE CASE

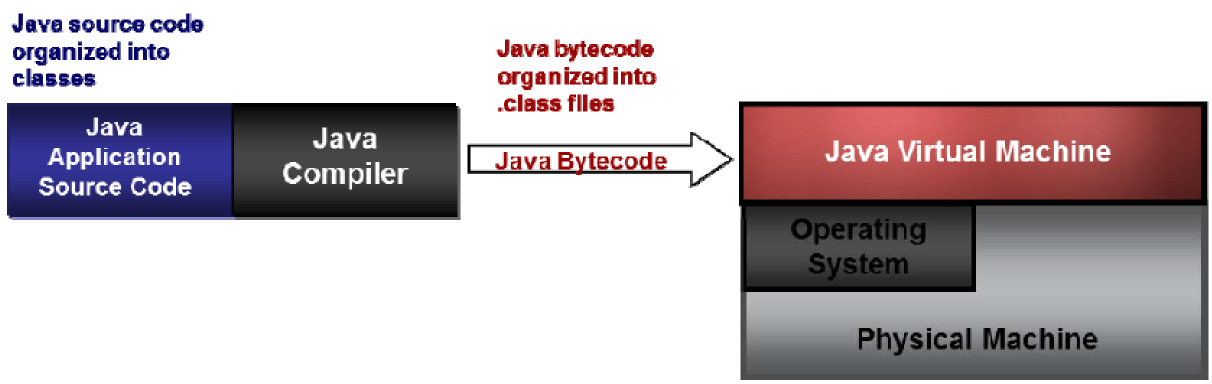
A. Technology Background

Since approximately 1960, computer programs have been written in human-readable programming languages, such as Fortran, Algol, C, and C++. A86 (1:10-29). Code written in these human-readable languages, called source code, is not directly executable by computer hardware. Instead, it must be converted to machine code in some way. As the name suggests, machine code uses the set of instructions that are understood and executed by a specific hardware processor. Different computer chips may execute different sets of rudimentary machine code instructions.

There are two traditional ways of converting source code into machine code. The first method uses a program called a compiler to convert an input file of source code to an output file of "native" machine code that can be executed later. A86 (1:58-60). This native machine code file can then be executed on a separate platform that does not have the source code or compiler. The second method uses a different kind of program called an interpreter. An interpreter processes source code instructions from an input file and executes corresponding machine instructions step-by-step, based on the source code instructions. A86 (1:55-58).

Unlike a compiler, an interpreter continues to run as the source code program is interpreted. In other words, the source code and interpreter must be present on any platform where the program is interpreted.

In the case of the Java platform, there is both compilation and interpretation. Java source code programs can be first compiled into an intermediate executable form called bytecode that is stored in class files. A86 (1:43-45). An interpreter, the Java virtual machine, is then used to interpret and execute Java bytecode. A86 (1:43-47). The Java virtual machine has specific functions that allow separately compiled class files to be incrementally loaded, verified, and then executed on any hardware platform that is equipped with the Java virtual machine. For example, as illustrated below, the Java virtual machine running as a program on a physical machine can receive the virtual machine instructions in the form of bytecodes and interpret the virtual machine instructions appropriately for the associated hardware of the physical machine:



An advantage of utilizing this approach is flexibility. The virtual machine instructions (*e.g.*, bytecodes) may be run, unmodified, on any computer system that has a virtual machine implementation (*e.g.*, a Java virtual machine). A86 (1:47-51). This is sometimes referred to as "write once, run anywhere," because a programmer only needs to write the source code once, and it can run anywhere.

While this approach provides flexibility, the interpretation of virtual machine instructions by a virtual machine can be less efficient than simply executing native code. The inventors of the '205 Patent, Lars Bak and Robert Griesemer, devised inventive ways of speeding up execution by a virtual machine. A86 (2:35-37).

B. The Invention Of The '205 Patent

The inventors of the '205 Patent recognized that the operation of the virtual machine may be augmented if, during interpretation, some native instructions are executed instead of interpreting virtual machine instructions (*e.g.*, bytecodes). A86 (2:35-40). This is best seen with respect to exemplary Figures 3 and 5 of the '205 Patent.

Figure 3 (below) illustrates Java source code 101 being complied into virtual machine instructions (*e.g.*, bytecodes) that are stored in a class file 105. These bytecodes are then received by the Java virtual machine 107, *i.e.*, an interpreter.

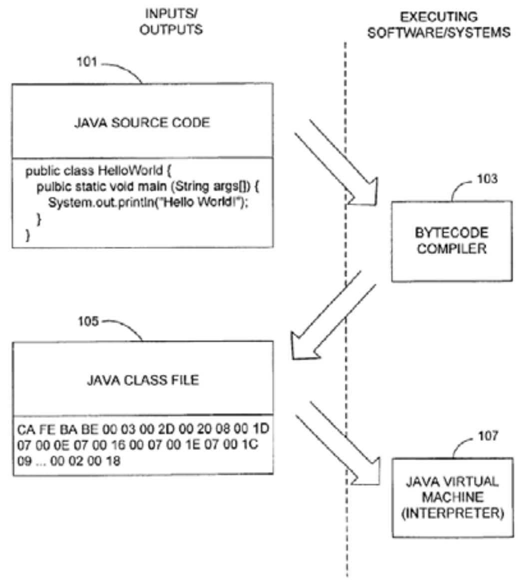


FIG. 3

A75. Figure 5 (below) at box 301 also illustrates the reception of bytecodes—*e.g.*, bytecodes 2-5—by a virtual machine, such as Java virtual machine 107.

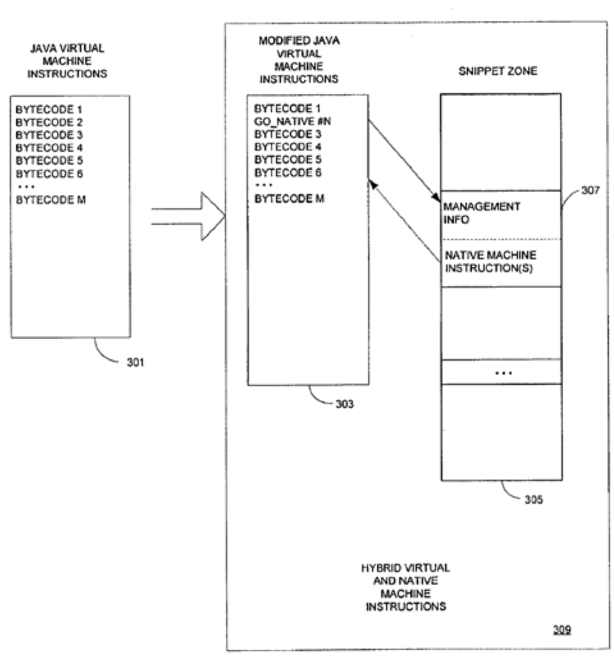


FIG. 5

A77. However, rather than simply interpreting bytecodes 2-5 for execution, the virtual machine of Figure 5 generates a new virtual machine instruction GO_NATIVE shown in box 303. The GO_NATIVE instruction includes a pointer or index #N to snippet 307 that includes native machine instructions corresponding to the operations of bytecodes 2-5, where the snippet can be executed instead of bytecodes 2-5. A89 (7:63-8:67). Then, the interpreter continues with the execution of bytecode 6 as if no snippet existed. A89 (8:33-37).

In this manner, the virtual machine generates the GO_NATIVE instruction to cause native machine instructions to be executed instead of the virtual machine instructions of bytecodes 2-5. This is advantageous, because the native machine instructions may achieve faster execution speed than the interpretation of virtual machine instructions. A86 (1:60-67, 2:35-40).

Claim 1 provides:

> 1. In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:
>
> receiving a first virtual machine instruction;
>
> generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction; and
>
> executing said new virtual machine instruction instead of said first virtual machine instruction.

A92 (13:44-53).

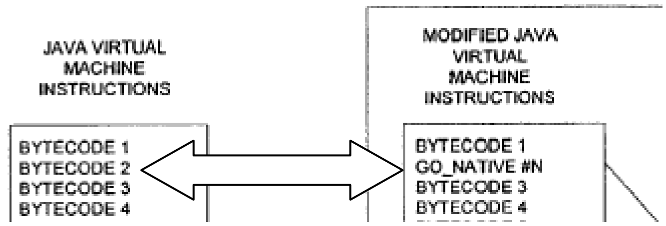
The example of Figure 5 is encompassed by claim 1: receiving "a first virtual machine instruction" (*e.g.*, a bytecode), generating "a new virtual machine instruction" (*e.g.*, a GO_NATIVE instruction) that represents or references one or more native instructions, and "executing said new virtual machine instruction instead of said first virtual machine instruction."

Exemplary dependent claim 2, and in particular, the feature of "overwriting," is critical to issues presented in this appeal. Claim 2 is set forth below:

> 2. The method of claim 1, further comprising overwriting a selected virtual machine instruction with a new virtual machine instruction, the new virtual machine instruction specifying execution of the at least one native machine instruction.

A92 (13:54-58).

In the example of Figure 5 (a portion of which is annotated and reproduced below), the virtual machine overwrites bytecode 2 with the new virtual machine instruction GO_NATIVE, which includes a pointer or index #N to specify execution of native machine instructions of snippet 307. A77; A89 (7:63-8:67). Indeed, Figure 5 shows the GO_NATIVE instruction ***written directly over*** the original bytecode 2 instruction in the modified Java virtual machine instructions box. A77; A89 (7:63-8:67).



Further, as described in an example in the specification (and claimed in dependent claim 4), the original virtual machine instruction (*e.g.*, bytecode 2) can be stored in a snippet zone prior to being overwritten so that the original virtual machine instruction may be regenerated or restored. A72 (Abstract); A89 (8:38-53); A92 (14:1-2). In other words, because an original virtual machine instruction (*e.g.*, bytecode 2) is literally written over by a new virtual machine instruction (*e.g.*, the GO_NATIVE instruction), the original virtual machine instruction must be stored elsewhere *prior* to overwriting if it is desired to be restored.

Aside from claims 2-4, new claims 15, 16, and 18-21 recite “overwriting.” All of these claims will be collectively called the “overwriting” claims. Only two claims on appeal—claims 1 and 8—do not recite “overwriting.”

C. Magnusson

Magnusson is an article that describes work on an instruction-level simulator used to develop and analyze computer architectures and system software. “Instruction-level simulation allows a program written for a particular machine to be executed on a dissimilar machine.” A481. Specifically, in Magnusson, the example of “a program written for a particular machine” is an “M88100” program

written for a Motorola-compatible microprocessor (the "target machine"). A486. Magnusson discloses that execution of the M88100 program may be simulated on a "host" machine employing "SPARC code," *e.g.*, a Sun SPARC microprocessor (the "dissimilar machine" of Magnusson's example). A486. The simulation of M88100 program execution is performed by a program, *i.e.*, a simulator, running on the host machine.

Although not entirely clear, Magnusson appears to describe a "partial translation" that combines (1) translation of some target machine instructions (*e.g.*, M88100 instructions) into an intermediate format, which is interpreted at the host machine, with (2) translation of some target machine instructions into native machine code (*e.g.*, SPARC code), which is executed by the host machine. A486.

Magnusson describes that an instruction entitled "TRANSLATED" is introduced and points to the native code:

> There are two ways of combining the threaded code model with block translation. Either the internal format includes a direct pointer to the compiled block, or *we introduce a new instruction, TRANSLATED*. This instruction takes as a parameter a pointer to the translated block, and handles generic entry/exit issues.

A487 (emphasis added).

As discussed further below (*see infra* Section III), Magnusson lacks working examples and is filled with discrepancies. As best understood, Magnusson's Figure 4 (annotated and reproduced below) appears to show the translation of the

target machine instructions (*e.g.*, M88100 instructions) into intermediate format instructions labeled "interpreted program" (shown in a rectangular box at the left of Figure 4). A486. As also best understood from the quote above, the TRANSLATED instruction is "introduced" into the intermediate format instruction stream of the "interpreted program."

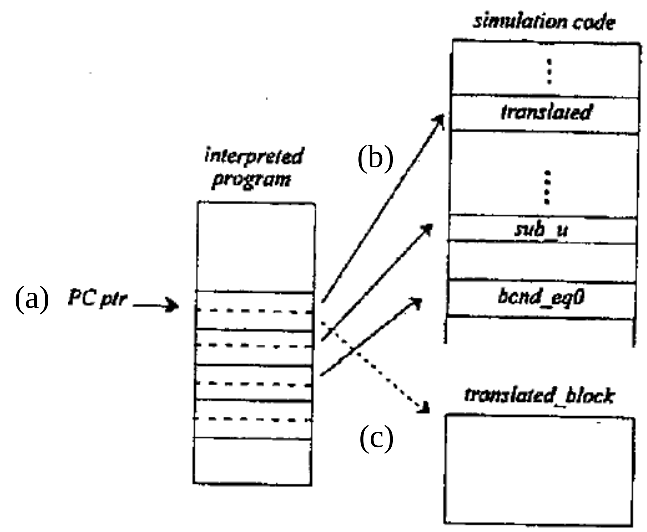


Figure 4 - Partial translation

Figure 4 of Magnusson illustrates that a "PC ptr" is at a location in the "interpreted program" in which an intermediate format instruction resides (annotated here as (a)). A486. When the execution of the "interpreted program" reaches this instruction, the program appears to jump to the corresponding service routine in the "simulation code" (annotated as (b)), which points to the translated block of native SPARC code (annotated as (c)). A486. The simulation code run by the simulator has "*translated*" as shown in Figure 4.

D. The Reexamination Proceeding In The Patent Office

On February 17, 2011, Google initiated an *inter partes* reexamination of the ’205 Patent of original claims 1-4 and 8. A100-A101. Google proposed that claims 1-4 and 8 should be rejected under 16 separate grounds—eight grounds of anticipation and eight grounds of obviousness. A139-A143. Magnusson was the eighth proposed anticipation rejection. A141. The Examiner found 14 of the 16 grounds to be deficient for claims 1-4 and 8 but rejected original claims 1-4 and 8 as anticipated by Magnusson and only claim 8 as anticipated by Tarau in a first Office Action dated August 19, 2011. A767-A775; A790-A810.

In response, Oracle explained that Magnusson was not enabled for the inventions of claims 1-4 and 8.[1] A828-A832. In addition, Oracle asserted that Magnusson failed to disclose the recited “overwriting” of claims 2-4. A832-A833. Finally, Oracle presented new “overwriting” claims 15-21 and explained that these claims were also not anticipated by Magnusson. A833.

Oracle supported its analysis with the testimony of its expert, Dr. Benjamin Goldberg. A836-A844; *see also* A1027-A1034. Google, for its part, responded that Magnusson was enabled and disclosed overwriting with testimony from its expert, Dr. Thomas Conte. A853-A899. The reexamination then proceeded to a

[1] The rejection based on Tarau was overcome by an amendment to claim 8. A977.

Right of Appeal Notice dated November 21, 2012. A13-A15. The Examiner maintained his anticipation rejection based on Magnusson for original claims 1-4 and 8 and newly added claims 15, 16, and 18-21 (with new claim 17 having been cancelled). A18. Oracle timely appealed to the Board. A1165.

In its Decision dated November 27, 2013, the Board first addressed the claims that recited the term "overwriting," *i.e.*, claims 2-4, 15, 16, and 18-21. The Board began by construing the term "overwriting." A7-A8. In three short paragraphs, the Board reviewed the specification of the '205 Patent and concluded "overwriting" does *not* mean writing over:

> We therefore interpret the term "overwriting" as the act of replacing some information in a computer file with new information, rather than literally writing over an existing information.

A8. This explicit claim construction of "overwriting" was not previously proffered in any paper during the reexamination, nor was it raised during the Board hearing. A1296-A1329.

Based on its claim construction, the Board then summarily concluded—in two sentences—that Magnusson anticipated the overwriting claims:

> We find that Magnusson discloses the use of intermediate code instructions for an emulated processor and then jumping to a native code (SPARC) block upon encountering a TRANSLATED instruction. Under the broad but reasonable interpretation of "overwriting" previously [set] forth, we find this process replaces instructions within the intermediate code with new

> information and therefore find that Magnusson discloses "overwriting."

A9.

Finally, the Board found that Magnusson was an enabling reference with equally sparse analysis: "We have similarly considered the record and we find that the arguments and evidence submitted by Owner are not sufficient to overcome the presumption of enablement relied upon by the Examiner." A11. The Board then sustained the anticipation rejection based on Magnusson for not only the "overwriting" claims 2-4, 15, 16, and 18-21, but also claims 1 and 8. A11.

SUMMARY OF ARGUMENT

I. The Board erred in construing "overwriting" in claims 2-4, 15, 16, and 18-21 as an act of replacing some information in a computer file with new information, rather than literally writing over an existing information.

First, the Board's claim construction flies in the face of the plain language of the claim term "overwriting." "Overwriting" literally means writing over. Yet, the Board construed "overwriting" *not* to mean writing over. It reached this illogical construction by ignoring the actual claim wording and skipping directly to the specification.

Second, the Board's claim construction is based on an erroneous reading of the specification of the '205 Patent. The specification is, in fact, consistent with "overwriting" as meaning writing over. Figure 5 of the '205 Patent shows a virtual

machine instruction ("BYTECODE 2") being literally overwritten by a new virtual machine instruction ("GO_NATIVE #N") as shown annotated below.

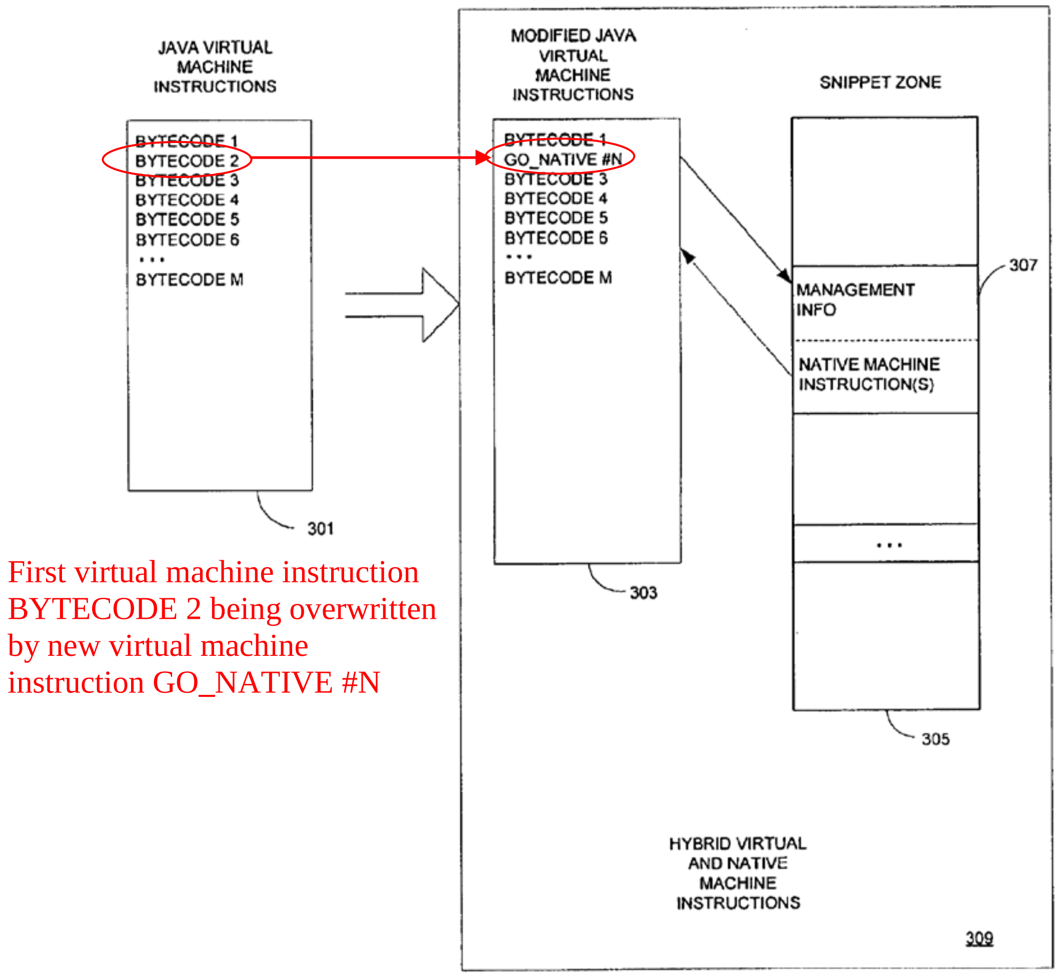


FIG. 5

A77. The specification expressly links the illustrated operation to "overwriting" by explaining that the "interpreter generates modified Java virtual machine instructions 303 by *overwriting bytecode 2 with a go_native virtual machine instruction*." A89 (8:2-5) (emphasis added). While the native instructions

Case: 14-1351 Document: 20 Page: 26 Filed: 05/13/2014

referenced by the GO_NATIVE instruction perform the same operations as if bytecodes 2-5 were interpreted, only bytecode 2—and *not* bytecodes 3-5—is overwritten with the GO_NATIVE instruction as shown.

Third, the Board's construction is inconsistent with other claims in the '205 Patent. The "overwriting" of exemplary dependent claim 2 is differentiated from the broader limitations of independent claim 1. Specifically, claim 2 requires a new virtual machine instruction (*e.g.*, a GO_NATIVE instruction) to overwrite a selected virtual machine instruction (*e.g.*, bytecode 2), whereas claim 1 simply requires that the new virtual machine instruction (*e.g.*, a GO_NATIVE instruction) be "executed instead of" one or more virtual machine instructions (*e.g.*, bytecodes 2-5). Construing "overwriting" as the Board has done makes claims 1 and 2 redundant.

II. The Board erred in affirming the Examiner's rejection of claims 2-4, 15, 16, and 18-21 as anticipated by Magnusson when Magnusson does not expressly or inherently disclose any "overwriting" as properly construed.

First, Figure 4 of Magnusson—upon which Google primarily relies for meeting "overwriting"—does not show "overwriting." According to Google, the "interpreted program" box in Figure 4 has virtual machine instructions, but Figure 4 does not show any instruction being overwritten in that box.

Second, Magnusson's statement that "we introduce a new instruction, TRANSLATED" cannot be relied upon for "overwriting," because to "introduce" does not expressly or inherently mean to "overwrite" as required by anticipation.

Third, Google's attempts to stitch together various portions of Magnusson are undermined by the fact that Figure 4 does not show any TRANSLATED instruction overwriting a virtual machine instruction. Given this deficiency, Google appears to be relying on inherent anticipation, but its showing of inherency is deficient. Google has only asserted *possibilities* for how the TRANSLATED instruction is introduced.

Fourth, Magnusson's routines labeled as "exit_0" and "exit_d" do not disclose "overwriting" as claimed. Google relies heavily on expert testimony to either supplement Magnusson's disclosure, which is improper for express anticipation, or to speculate about Magnusson's operation, which is insufficient for inherent anticipation.

III. The Board erred in affirming the Examiner's finding that Magnusson is an enabling reference so as to anticipate claims 1-4, 8, 15, 16, and 18-21.

First, the Board failed to recognize that Magnusson does not provide an enabling disclosure to a key component of the anticipation rejection, *i.e.*, how Magnusson's TRANSLATED instruction is to be introduced.

Second, the Board performed an inappropriate and erroneous comparison of certain portions of Magnusson with the specification of the ’205 Patent in view of this Court’s decision in *In re Morsa*, 713 F.3d 104 (Fed. Cir. 2013).

Third, the Board utilized an inappropriate level of skill in the art—that of an expert—to evidence enablement.

STANDARD OF REVIEW

The Board’s factual findings are reviewed for substantial evidence and the Board’s legal conclusions are reviewed de novo. *In re Kotzab*, 217 F.3d 1365, 1369 (Fed. Cir. 2000); *In re Gartside*, 203 F.3d 1305, 1316 (Fed. Cir. 2000). “Although the PTO gives claims the broadest reasonable interpretation consistent with the written description, claim construction by the PTO is a question of law that [this Court] review[s] *de novo*, just as [the Court] review[s] claim construction by a district court.” *In re Baker Hughes Inc.*, 215 F.3d 1297, 1301 (Fed. Cir. 2000) (citations omitted); *see also Lighting Ballast Control LLC v. Philips Elecs. N. Am. Corp.*, 744 F.3d 1272, 1276-77 (Fed. Cir. 2014) (en banc).

To anticipate, every claim limitation must be either expressly or inherently disclosed in a single, enabled prior art reference. *Transclean Corp. v. Bridgewood Servs., Inc.*, 290 F.3d 1364, 1370 (Fed. Cir. 2002); *Glaxo Inc. v. Novopharm Ltd.*, 52 F.3d 1043, 1047 (Fed. Cir. 1995). Anticipation is a question of fact. *Cheese Sys., Inc. v. Tetra Pak Cheese & Powder Sys., Inc.*, 725 F.3d 1341, 1347 (Fed. Cir.

2013). Enablement of prior art is a question of law, but is based on the underlying factual findings. *In re Morsa*, 713 F.3d at 109. Although the "substantial evidence" standard of review requires deference to the Board's factual findings, the Court can reverse the Board for incorrectly analyzing the factual findings. *Smith & Nephew, Inc. v. Rea*, 721 F.3d 1371, 1380 (Fed. Cir. 2013).

ARGUMENT

I. THE BOARD ERRED IN CONSTRUING "OVERWRITING"

The Board construed "overwriting" as the act of replacing some information in a computer file with new information, rather than literally writing over existing information, and affirmed the rejection of claims 2-4, 15, 16 and 18-21. A8; A12. The Board erred for at least three reasons.

A. The Claim Limitation "Overwriting" Plainly Means Writing Over

First, the Board did not consider the claim language when it construed "overwriting" *not* to mean overwriting. The Board explained that there is "a 'heavy presumption' that a claim term carries its ordinary and customary meaning," but it then skipped over the claim language and went straight to the specification:

> Owner contends that Magnusson fails to disclose "overwriting" (App. Br. 11-12); however, Owner does not point to any disclosure in the '205 patent that sets forth a clear definition for "overwriting."

> *Consequently, we must look to the disclosure of the ’205 patent to determine a proper definition for “overwriting.”*

A7 (emphasis added).

Claim construction, however, “‘begins and ends in all cases with the actual words of the claim.’” *Becton, Dickinson & Co. v. Tyco Healthcare Grp., LP*, 616 F.3d 1249, 1254 (Fed. Cir. 2010) (quoting *Renishaw PLC v. Marposs Societa’ per Azioni*, 158 F.3d 1243, 1248 (Fed. Cir. 1998)). Indeed, in the en banc decision of *Phillips v. AWH Corp.*, 415 F.3d 1303 (Fed. Cir. 2005) (en banc), the Court explained that “[q]uite apart from the written description and the prosecution history, the claims themselves provide substantial guidance as to the meaning of particular claim terms.” *Id.* at 1314. Here, the claim term “overwriting” could not be clearer. “Overwriting” literally means writing over. By ignoring the express claim language, the Board reached the illogical construction that “overwriting” means *not* writing over.

In re Suitco Surface, Inc., 603 F.3d 1255 (Fed. Cir. 2010) is instructive. There, the express claim language was a “material for finishing the top surface of the floor.” *Id*. at 1260. The Board’s construction, however, did not require finishing the top layer of a surface, as set forth in the claim language. *Id*. This Court vacated the Board’s decision, because the Board had “ignored [the] reality”

of the claim language. *Id*. In this present appeal, the reality of "overwriting" is writing over.

Furthermore, the Board emphasized in its decision that there is a "'heavy presumption' that a claim term carries its ordinary and customary meaning." A7; *see also Ancora Techs., Inc., v. Apple, Inc.*, 744 F.3d 732, 734 (Fed. Cir. 2014) ("A claim term should be given its ordinary meaning in the pertinent context, unless the patentee has made clear its adoption of a different definition or otherwise disclaimed that meaning."). The Board looked for an express definition in the specification but, having found none, it never paused to consider the ordinary and customary meaning of the claim term "overwriting." A7. Had it done so, it would not have reached its illogical claim construction of "overwriting" as not meaning writing over. One of ordinary skill in the art would not have understood "overwriting" to mean anything other than writing over.

Indeed, throughout the reexamination, neither the Examiner nor Google provided any explicit definition of "overwriting," but both the Examiner and Google appeared to construe "overwriting" as literally writing over existing information with new information (*e.g.*, literally writing over a virtual machine instruction with a new virtual machine instruction in the context of claim 2). *See, e.g.*, A1050 (Google stating that "Figure 4 of the Magnusson reference illustrates the introduction of a new instruction into the intermediate code; this introduction

must overwrite existing code, else there would be no way to jump to the translated block."); A44 (Examiner in agreement). Even during the Board hearing, there was never any discussion of another definition of "overwriting." A1296-A1329. The first appearance of the new construction was in the Board's decision, where the Board erroneously skipped over the plain claim language. *See, e.g., Suitco Surface*, 603 F.3d at 1260 (vacating the Board's claim construction in a reexamination for "not reasonably reflect[ing] the plain language" of the claim). Had the Board considered the claim language and followed its own "heavy presumption" that a claim term carries its ordinary and customary meaning, it would not have erred.

B. The Specification Is Consistent With The Plain Meaning of "Overwriting"

Second, the Board erred by misunderstanding the specification of the '205 Patent. The specification is consistent with "overwriting" as writing over. For example, overwriting is illustrated in Figure 5 and described in the specification at column 8, lines 2-16. A77; A89 (8:2-16). Figure 5 (annotated below) plainly shows that original virtual machine instruction "BYTECODE 2" has been *written over* by a new virtual machine instruction "GO_NATIVE #N."

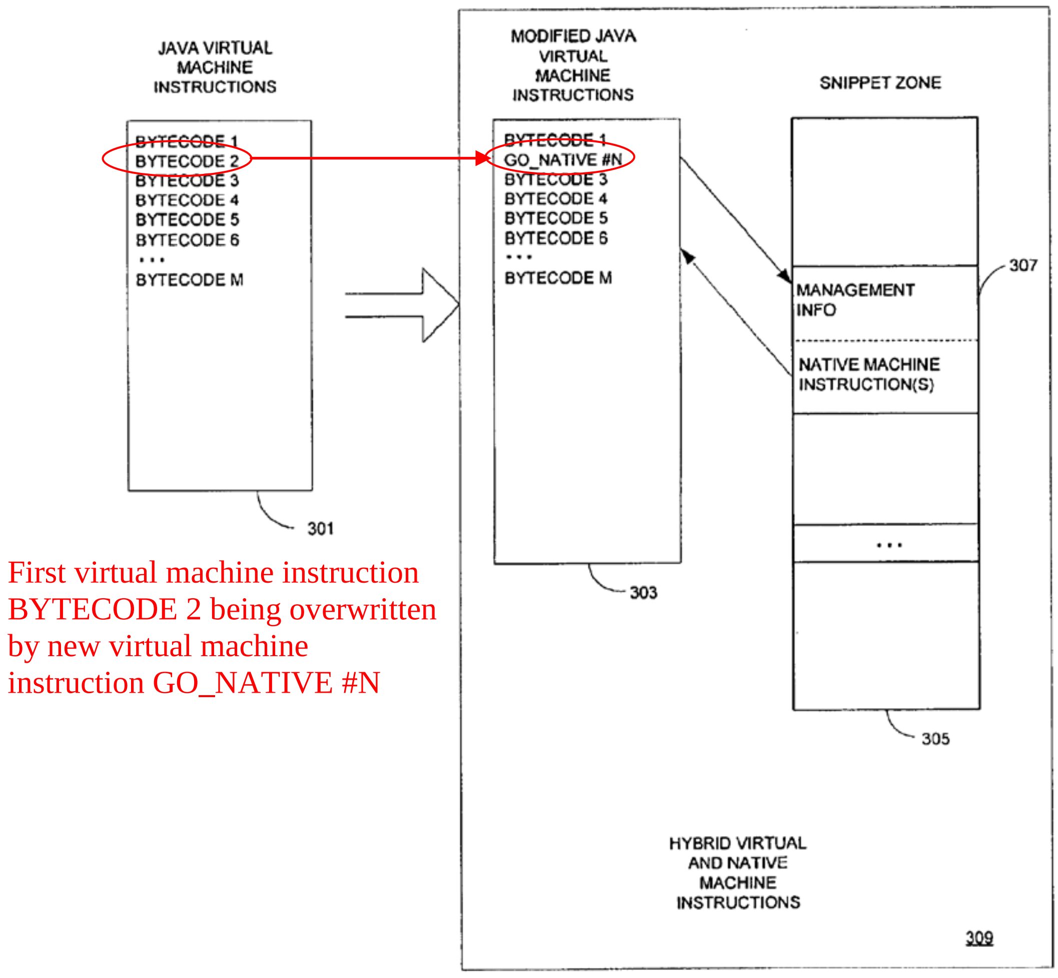


FIG. 5

A77. This operation shown in annotated Figure 5 is expressly linked to the term "overwriting," because the specification explains that the "interpreter generates modified Java virtual machine instructions 303 by *overwriting bytecode 2 with a go_native virtual machine instruction*." A89 (8:3-5) (emphasis added).

Yet the Board misread this description of "overwriting" in the specification. It stated:

Case: 14-1351 Document: 20 Page: 34 Filed: 05/13/2014

> We note that the explicit illustration within the '205 patent is that only bytecode 2 has been "overwritten" within modified Java virtual machine instructions 303, *despite the fact that bytecodes 3-5 are also being replaced by one or more native machine instructions*. *See* '205 patent, Fig. 5. We therefore interpret the term "overwriting" as the act of replacing some information in a computer file with new information, rather than literally writing over an existing information.

A8 (emphasis added).

The Board's analysis conflates the concept of "overwriting" with the operations performed by the referenced native instructions. As shown in Figure 5, *only* bytecode 2 is overwritten with the GO_NATIVE instruction, which in turn references native instructions that perform the operations of bytecode 2 and bytecodes 3-5. The specification does *not* describe that bytecodes 3-5 are overwritten. Native machine instructions are executed instead of bytecodes 3-5, *i.e.*, bytecodes 3-5 are skipped in favor of native machine instructions in snippet 307. However, there is nothing to suggest that bytecodes 3-5 are overwritten with new virtual machine instructions (*i.e.*, GO_NATIVE instructions). The Board thus confused the operations performed by the referenced native machine instructions—which are the same as bytecodes 2-5—with the concept of overwriting bytecode 2.

The conclusion that the specification discloses only overwriting bytecode 2—and not bytecodes 3-5—is further confirmed by other passages in the specification. The specification describes that the "original bytecode 2 which was

Case: 14-1351 Document: 20 Page: 35 Filed: 05/13/2014

overwritten by the go_native bytecode and also the original address of bytecode 2" can be stored within the management information of the snippet 307 "so that the original bytecode sequence may be restored when the snippet is removed." A89 (8:11-16). In context, the '205 patent explains:

> A snippet zone 305 stores snippets which include native machine instructions. As shown, the go_native bytecode includes a pointer or index to a snippet 307. Each snippet is a data block that includes two sections of which the first is management information and the second is a sequence of one or more native machine instructions. *The management information includes storage for the original bytecode 2 which was overwritten by the go_native bytecode and also the original address of bytecode 2 so that the original bytecode sequence may be restored when the snippet is removed*. Typically, the management information section of the snippet is of a fixed length so that the native machine instructions may be easily accessed by a fixed offset.

A89 (8:6-18) (emphasis added); *see also* A72 (Abstract); A87 (3:1-4); A90 (9:19-29); A78 (illustrating the virtual machine instruction is saved at step 405 prior to overwriting in step 409); A92 (14:1-2). In other words, because an original virtual machine instruction (*e.g.*, bytecode 2) is literally written over by a new virtual machine instruction (*e.g.*, the GO_NATIVE instruction), the original virtual machine instruction must be stored elsewhere *prior* to overwriting if it is desired to be restored.

Contrary to the Board's analysis, bytecodes 3-5 are not overwritten. There is no mention in the specification of the '205 Patent that these bytecodes 3-5 are stored for later regeneration as described specifically with bytecode 2. Again, snippet 307 includes the *native machine instructions* corresponding to virtual machine instructions of bytecodes 3-5. Notably, the '205 Patent was careful not to describe the execution of native machine instructions instead of bytecodes 3-5 as "overwriting." Only bytecode 2 is described in the specification with respect to the act of "overwriting" and storing for later regeneration and, thus, the specification further supports the plain meaning of "overwriting." *See Suitco Surface*, 603 F.3d at 1260 ("The broadest-construction rubric coupled with the term 'comprising' does not give the PTO an unfettered license to interpret claims to embrace anything remotely related to the claimed invention. Rather, claims should always be read in light of the specification and teachings in the underlying patent."). Moreover, the specification has not adopted a different definition or otherwise disclaimed the plain meaning of "overwriting." *Ancora*, 744 F.3d at 734.

C. Claim Differentiation Supports The Plain Meaning Of "Overwriting"

Third, the Board erred in failing to consider other claims in construing "overwriting." The *Phillips* Court explained that "the context in which a term is used in the asserted claim can be highly instructive" and that "[o]ther claims of the

patent in question, both asserted or unasserted, can also be valuable sources of enlightenment as to the meaning of a claim term." *Phillips*, 415 F.3d at 1314.

The "overwriting" of exemplary dependent claim 2 is differentiated from the broader limitations of independent claim 1. Claim 1 recites generating a new virtual machine instruction that can be executed instead of a first virtual machine instruction. A92 (13:44-53). For example, in Figure 5, the virtual machine generates a new GO_NATIVE virtual machine instruction that is executed instead of bytecodes 2-5. A77. Exemplary claim 2 further adds "overwriting" a selected virtual machine instruction with a new virtual machine instruction. A92 (13:54-58). In the example of Figure 5, bytecode 2 is overwritten with the GO_NATIVE instruction. A77.

Note that only bytecode 2 is overwritten with the GO_NATIVE instruction, even though the GO_NATIVE instruction is executed instead of bytecodes 2-5. This is precisely the difference in scope between claims 1 and 2 that the Board failed to appreciate: claim 2 requires a new virtual machine instruction (*e.g.*, a GO_NATIVE instruction) to overwrite a selected virtual machine instruction (*e.g.*, bytecode 2), whereas claim 1 simply requires that the new virtual machine instruction (*e.g.*, a GO_NATIVE instruction) be executed instead of one or more virtual machine instructions (*e.g.*, bytecodes 2-5). Thus, if "overwriting" is construed as something other than literally writing over, as the Board has done,

there is nothing that differentiates the specific requirement of overwriting in dependent claim 2 from the broader scope of independent claim 1. Such redundancy in claim scope—because claim 1 already recites "executed instead of"—violates the doctrine of claim differentiation, which is based on the "common sense notion that different words or phrases used in separate claims are presumed to indicate that the claims have different meanings and scope." *Karlin Tech., Inc. v. Surgical Dynamics, Inc.*, 177 F.3d 968, 971-72 (Fed. Cir. 1999).

Accordingly, the Board erred. It failed to apply the plain meaning of "overwriting" even though it noted the "heavy presumption" that a claim term carries its ordinary and customary meaning. A7. The Board misread the specification of the '205 Patent despite its clear discussion and illustration of "overwriting." A77; A89 (8:2-5). Finally, the Board failed to consider other claims that would have provide even further guidance. The plain claim language, the specification and others claims unequivocally establish that "overwriting" means literally writing over and, in the context of claim 2, overwriting a virtual machine instruction (*e.g.*, bytecode 2) with a new virtual machine instruction (*e.g.*, a GO_NATIVE instruction). Given that the Board based its affirmance of the anticipation rejection on an unreasonable construction that "overwriting" is *not* writing over, this Court should remand to conduct a new invalidity analysis using

the proper construction of "overwriting." *See, e.g.*, *Suitco Surface*, 603 F.3d at 1261.

II. MAGNUSSON DOES NOT DISCLOSE "OVERWRITING" UNDER THE PROPER CONSTRUCTION OF "OVERWRITING"

As discussed above, the Board incorrectly construed "overwriting." This error led it to incorrectly conclude—in two sentences—that Magnusson anticipates claims 2-4, 15, 16, and 18-21. A9. As also discussed above, Oracle, Google and the Examiner during the reexamination viewed (or appeared to view) "overwriting" as literally writing over existing information with new information. Thus, much of the record is directed to whether Magnusson discloses "overwriting" under its proper construction. Accordingly, if the Court wishes to address whether Magnusson anticipates claims 2-4, 15, 16, and 18-21 under the proper construction of "overwriting," the record demonstrates that Magnusson is deficient and fails to anticipate claims 2-4, 15, 16, and 18-21.

A. The "Interpreted Program" In Magnusson's Figure 4 Does Not Show "Overwriting"

During the reexamination, Google settled on the "interpreted program" of Magnusson's Figure 4 as meeting a "virtual machine instruction" required by exemplary claims 1 and 2. *See, e.g.*, A1223. Specifically, Magnusson discloses that target machine instructions (*i.e.*, M88100 instructions) are translated into

intermediate format instructions labeled “interpreted program” shown in a rectangular box at the left of Figure 4 (below and with a block arrow added).

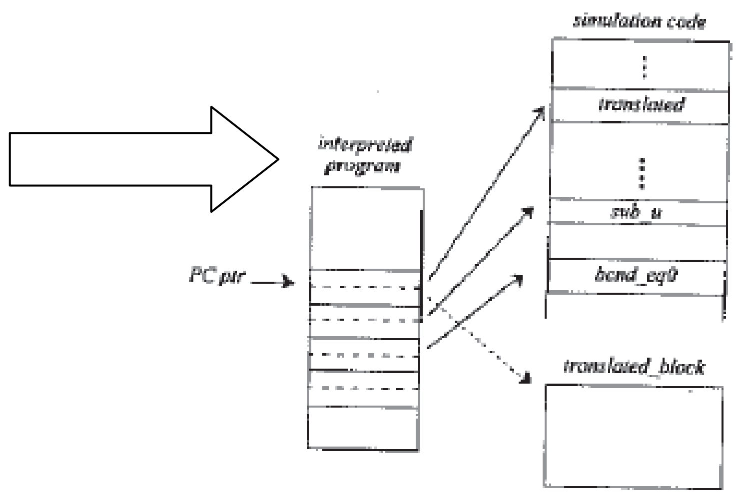


Figure 4 - Partial translation

A486.

Exemplary claim 2, however, requires *overwriting* a virtual machine instruction with a new virtual machine instruction. Thus, during the reexamination, Google sought to show that an intermediate format instruction in the “interpreted program” of Figure 4 was overwritten. *See, e.g.*, A1223-A1224. Google certainly could not point to the “interpreted program” as illustrated in Figure 4 above. There is nothing in the “interpreted program” as illustrated that shows an intermediate format instruction being overwritten. All that it shows are blocks of intermediate format instructions, which is insufficient for establishing anticipation. *See, e.g.*, *In re Skvorecz*, 580 F.3d 1262, 1268 (Fed. Cir. 2009)

("Anticipation cannot be found, as a matter of law, if any claimed element or limitation is not present in the reference.").

B. The TRANSLATED Instruction In Magnusson's Narrative Does Not Disclose "Overwriting"

Google accordingly sought to find the "overwritten" instruction in Magnusson's narrative, particularly the following passage:

> There are two ways of combining the threaded code model with block translation. Either the internal format includes a direct pointer to the compiled block, or *we introduce a new instruction, TRANSLATED*. This instruction takes as a parameter a pointer to the translated block, and handles generic entry/exit issues.

A487 (emphasis added); *see also* A862.

On its face, this passage does not disclose "overwriting" an intermediate format instruction in the "interpreted program" of Figure 4. It does not even mention "overwriting." Google apparently relies on the word "introduce" to mean "overwriting." However, to "introduce" is *not* the same as "overwriting." The ordinary meaning of to "introduce" is to "put into use" or "present." This is evident by how Magnusson is using the term in the passage quoted above. All that Magnusson is stating is that a new function or operation can be put into use—and nothing more.

But to anticipate, Magnusson must disclose every claim limitation, either explicitly or inherently, and that determination is reviewed for substantial

evidence. *Glaxo*, 52 F.3d at 1047; *Suitco Surface*, 603 F.3d at 1259. To "introduce" in the quote above does not provide evidence as to anticipating "overwriting," because to "introduce" neither explicitly discloses nor inherently requires "overwriting." *See* A1032 (¶14) ("[T]he ACP simply cites the portion of Magnusson that describes the introduction of the TRANSLATED instruction, but not 'overwriting' or how a person skilled in the art reading these portions would understand the introduction to mean 'overwriting.'"). Thus, this evidentiary showing upon which Google relies falls short of the "substantial evidence" required to affirm an anticipation rejection.

C. Even Stitching Isolated Sections Of Magnusson Together Does Not Disclose "Overwriting"

Given the deficiencies of the "interpreted program" illustrated in Figure 4 and Magnusson's narrative of the TRANSLATED instruction, Google has sought to stitch together various isolated portions of Magnusson and Figure 4 to establish "overwriting." Google primarily relies on the following passage describing Figure 4 in Magnusson:

> Figure 4 illustrates how of [sic] interpreting an intermediate format can be combined with direct execution of generated code. Assume that we wish to translate a sequence [of] M88100 instructions to native SPARC code. The intermediate code is in a 64-bit format, where the first 32 bits points to the code that simulates the corresponding instruction. The second 32

> bits contain parameters for the instruction. Each 88k instruction is translated to this format.
>
> When execution of the program reaches the first instruction in the block, the service routine being jumped to is actually a run-time generated block of SPARC code.

A486.

Magnusson appears to describe that M88100 instructions ("target code") are compiled to 64-bit-format intermediate format instructions ("interpreted program" in Figure 4). As best understood from the above quote, the first 32 bits of a given intermediate format instruction points to code ("*translated*") that simulates the TRANSLATED instruction and the second 32 bits point to the run-time generated block of SPARC code ("*translated_block*" in Figure 4).

Google has spent much of the reexamination trying to stitch this discussion with other passages and figures in Magnusson such as a discussion of a threaded code model in Figure 3, a lone sentence of runtime operations on page 6 (A484), and the narrative discussion of the TRANSLATED instruction at page 9 (A487). *See, e.g.*, A861-A865. Based on this stitching, Google repeatedly concludes that a TRANSLATED instruction overwrites an intermediate format instruction in the "interpreted program" of Figure 4:

> Accordingly, Magnusson's Figure 4 depiction of the TRANSLATED instruction in the threaded code instructions (i.e., the "interpreted program") is the result of the simulator's overwriting—at runtime—an original instruction with the newly-generated TRANSLATED instruction.

Case: 14-1351 Document: 20 Page: 44 Filed: 05/13/2014

A865.

> Figure 4 of the Magnusson reference illustrates the introduction of a new instruction into the intermediate code; this introduction must overwrite existing code, else there would be no way to jump to the translated block.

A1050.

However, the conclusion—and thus Google's entire analysis—is belied by Figure 4 of Magnusson. Figure 4 does *not* provide a "depiction" of the TRANSLATED instruction in the "interpreted program" nor does it "illustrate" the introduction of the TRANSLATED instruction as quoted above. Instead, the "interpreted program" of Figure 4 is blank as shown below with a block arrow.

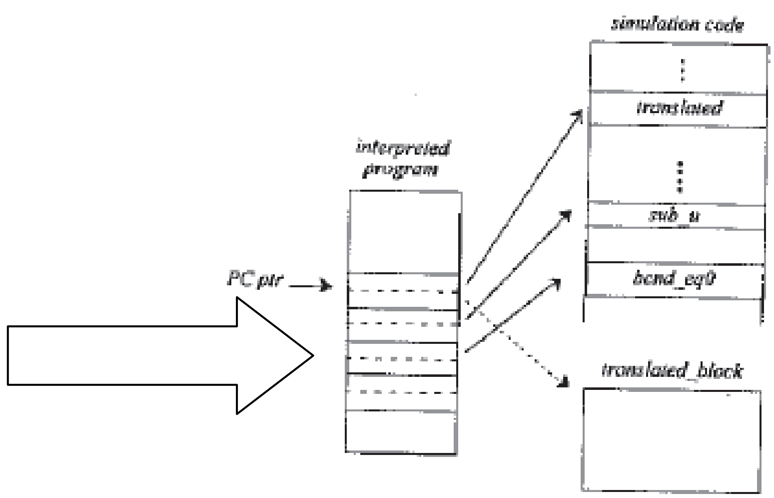


Figure 4 - Partial translation

A486.

If Magnusson actually disclosed "overwriting" by the TRANSLATED instruction in Figure 4 under Google's convoluted analysis, there would have been

some written indication in the "interpreted program," such as the term TRANSLATED, just as "*translated*" is written in the "simulation" code. The conspicuous absence of TRANSLATED in Figure 4 directly refutes Google's assertions of the "depiction" or "illustration" of the TRANSLATED instruction and thus undermines Google's analysis.

It is important to note that Google is not relying on the "simulation code" in Figure 4 to establish "overwriting" as claimed. The "simulation code" is the code of the simulator and is shown in a rectangular box on the right side of Figure 4 (below and with a block arrow added).

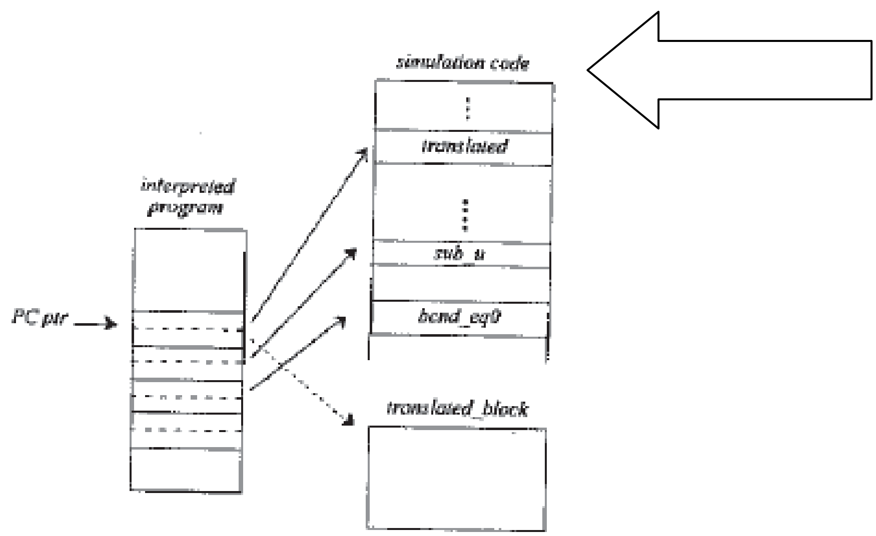


Figure 4 - Partial translation

A486.

While the simulation code includes a code "*translated*," the "simulation code" is *not* the intermediate format instructions of the "interpreted program" at the left of Figure 4. Thus, Google does not rely on the "*translated*" code in the

"simulation code" for "overwriting, but rather that a TRANSLATED instruction has overwritten an intermediate format instruction in the "interpreted program" at Figure 4. *See, e.g.*, A1223-A1224; A1319. As discussed above, there is no such "depiction" or "illustrat[ion]" of the TRANSLATED instruction in the "interpreted program" of Figure 4.

In view of the lack of express disclosure, Google appears to assert *inherent* anticipation of "overwriting." For example, in its brief to the Board, Google made the following statement: "Magnusson's 'introduction' of a new instruction must necessarily 'overwrite' the original instruction; the code Magnusson provides makes this an inescapable conclusion." A1224; *see also* A1050 ("[T]his introduction must overwrite existing code, else there would be no way to jump to the translated block.").

The conclusion of "overwriting" from "introduce" is certainly not "inescapable" as Google asserts. Figure 4, in fact, supports Oracle's understanding of Magnusson's "introduce," as meaning to "put into use" or "present" and not the "overwriting" as claimed and discussed in '205 Patent. In the '205 Patent, a virtual machine is presented with a stream of bytecode instructions. It can either interpret them or overwrite a given bytecode instruction with a new instruction. A1032(¶15). In contrast, Magnusson's system as best understood translates source code (*e.g.*, M88100 code) into intermediate format instructions. Given this act of

translation, Dr. Goldberg—Oracle's expert—testified that there would be no reason for Magnusson to translate source code into an intermediate format instruction just to then have it overwritten with a TRANSLATED instruction. A1032 (¶15). Rather, the TRANSLATED instruction is "introduced" or simply presented in the intermediate format instruction stream without overwriting any intermediate format instructions which have been translated from source code.

Thus, in view of the record, Google has merely asserted possibilities for how Magnusson's TRANSLATED instruction is introduced, thereby defeating Google's assertion of inherent anticipation. *See, e.g., Finnigan Corp. v. Int'l Trade Comm'n*, 180 F.3d 1354, 1366 (Fed. Cir. 1999) ("The mere possibility that Figure 2 might be understood by one of skill in the art to disclose nonresonance ejection is insufficient to show that it is inherently disclosed therein."). Google has failed to establish that the TRANSLATED instruction *necessarily* overwrites an intermediate format instruction in the "interpreted program" of Figure 4.

Accordingly, when one considers Figure 4 of Magnusson, its narrative as to the TRANSLATED instruction, and the remainder of the isolated passages cited by Google, it is clear that Google (and the Examiner) failed during the reexamination to show that the TRANSLATED instruction expressly or necessarily overwrites an original virtual machine instruction. It was error for the Board to affirm the rejection of claims 2-4, 15, 16, and 18-21 as anticipated by

Case: 14-1351 Document: 20 Page: 48 Filed: 05/13/2014

Magnusson. *See, e.g.*, *In re Robertson*, 169 F.3d 743, 745 (Fed. Cir. 1999) ("Anticipation . . . requires that each and every element as set forth in the claim is found, either expressly or inherently described, in a single prior art reference.") (internal quotation marks omitted).

D. The "Exit_0" Routine Does Not Disclose "Overwriting"

Although the Board did not address arguments made during the reexamination by Google (and adopted by the Examiner) based on Magnusson's "exit_0" routine, given the incorrect claim construction, Oracle further describes how this stretched assertion fails to disclose the "overwriting" limitation. In particular, Google asserts that Magnusson's "exit_0" routine discloses the TRANSLATED instruction overwriting a virtual machine instruction, thereby disclosing "overwriting." A42-A44. Google is wrong.

First, the discussion of the "exit_0" routine does not disclose any overwriting on its face. The "exit_0" code reads as follows:

```
rOP = upper<CONST>            ; load rOP with description of first
rOP |= lower<CONST>           ;  instruction ("r7 = [r6]")
br <sim_read>                 ; and branch to service routine
```

A489. Magnusson further describes the operation of the "exit_0" routine as follows:

> Exit_0 dispatches the first instruction of the block as if it had been an interpreted instruction. It does this by loading the corresponding rOP value and branching to the interpretation code.

A489. The absence of overwriting is clear. Nowhere in either the routine itself or Magnusson's description of the routine is there *any* disclosure (express or inherent) of overwriting an intermediate format instruction in the "interpreted program" with the TRANSLATED instruction.

Indeed, the "exit_0" routine does not change the intermediate format instruction at all. Rather, what the "exit_0" routine discloses (as best understood) is loading an instruction "description" and then branching off to a service routine. Specifically, this routine describes loading an instruction description from <CONST> and then branching to the "sim_read" service routine. This means that the "exit_0" routine simply branches to the "simulation code" that simulates the corresponding M88100 target instruction to be interpreted, according to Dr. Goldberg. That's all—no overwriting. A843-A844 (¶¶27-29); A1031-A1033 (¶¶14-16).

Facing a lack of express disclosure, Google asserts that the "overwriting" is implied from the "exit_0" routine. Google provides the following unsupported and overreaching gloss on the "exit_0" routine:

> It is clear from Magnusson's description of the "exit_0" code that the original virtual machine instruction is stored in the <CONST> constant value at block translation time. *See* Conte Declaration ¶ 30. This is why Magnusson states that the value of <CONST> is only known at compile time. *See* Conte Declaration ¶ 30. The reason that Magnusson stores the original instruction

> in the <CONST> constant is because, as discussed above, the TRANSLATED instruction overwrites the original virtual machine instruction within the threaded code instruction sequence (i.e., the "interpreted program" of Figure 4). *See* Conte Declaration ¶ 30.

A866.

Google is again wrong. Here, Google has imagined that an "original virtual machine instruction" is stored in the <CONST> value and that such storage reflects that the TRANSLATED instruction has overwritten that instruction. This is speculation, as there is no discussion linking the "exit_0" routine with the TRANSLATED instruction. Even if an instruction description is stored in <CONST>, such storage is not *necessarily* due to overwriting. Dr. Goldberg testified that <CONST> would contain information about an intermediate format instruction because, in the intermediate format instruction stream, the TRANSLATED instruction was used instead of the intermediate format instruction. A843-A844 (¶¶27-29); A1031-A1033 (¶¶14-16). In other words, the TRANSLATED instruction is introduced instead of an intermediate format instruction (not over it) and, upon completion of translation, the intermediate format instruction is interpreted.

Google has, at best, merely argued possibilities that the TRANSLATED instruction could have, should have, or would have overwritten an intermediate format instruction. This is precisely the kind of speculation that the law of

inherency prohibits in the context of anticipation. *See, e.g., In re Robertson*, 169 F.3d at 745.

E. The "Exit_d" Routine Does Not Disclose "Overwriting"

Although the Board did not address arguments made during the reexamination by Google (and adopted by the Examiner) based on Magnusson's "exit_d" routine, given the incorrect claim construction, Oracle further describes how this stretched assertion also fails to disclose the "overwriting" limitation. Google asserts that Magnusson's "exit_d" routine discloses the TRANSLATED instruction overwriting a virtual machine instruction, thereby disclosing "overwriting." A42-A44.

The "exit_d" code reads as follows:

```
call <dealloc_trans>       ; routine needs no params (uses rPC)
[rPC] = 0;                 ; invalidate self
rOP = 0;
br rCODE                   ; force new translation
```

A489. Magnusson describes what the "exit_d" routine does as follows:

> The deallocating exit (exit_d) calls a routine that deallocates the space used by the code. This routine uses rPC to determine which space to remove. It does not, however, overwrite the block, and therefore the block can still execute! When the routine returns, it invalidates the translation in the intermediate code, and dispatches opcode "0", which corresponds to *not_decoded*.

A489. Nowhere in the routine itself nor Magnusson's description of the routine is there an express or inherent disclosure of "overwriting" as recited. Rather, this

routine merely describes replacing the current instruction that rPC points to with "0"—not a *virtual machine instruction*—so that the current instruction is not executed, and then branching to rCODE to "force new translation." That's all—no overwriting. A843-A844 (¶¶27-29); A1031-A1033 (¶¶14-16).

Google once again provides an unsupported and overreaching gloss, but this time on the "exit_d" routine:

> The "[rPC] = 0; invalidate self" line in the above "exit_d" routine corresponds to Magnusson's description of "invalidat[ing] the translation in the intermediate code," and actually shows that the virtual machine instruction in the intermediate code is over-written with the value "0." *See* Conte Declaration ¶ 33. In other words, if the translated block was entered based on a TRANSLATED instruction, this line of code will essentially overwrite the TRANSLATED instruction with the value "0." *See* Conte Declaration ¶ 33.

A867.

Google is again wrong. Here, Google has reached at least two conclusions clearly unsupported by the "exit_d" routine itself and Magnusson's disclosure thereof. First, the value "0" is not an instruction or virtual instruction and therefore cannot be executed. Therefore, the value "0" is not the same as the "new virtual machine instruction" *required by claims at issue to be executed*. Indeed, what happens to the intermediate format code after the "force new translation" step in "exit_d" is unknown. Second, even were the TRANSLATED instruction overwritten with "0," this overwriting of the TRANSLATED instruction is

Case: 14-1351 Document: 20 Page: 53 Filed: 05/13/2014

irrelevant because the claim language requires that the TRANSLATED instruction, *i.e.*, the purported “new virtual machine instruction,” *do* the overwriting, not *be* itself overwritten. A843-844 (¶¶27-29); A1031-A1033 (¶¶14-16).

Furthermore, as described above, Google speculates that the “exit_0” routine stores an original virtual machine instruction in <CONST> that is overwritten by the TRANSLATED instruction. If that were the case, the “exit_d” routine would then overwrite the TRANSLATED instruction with the value stored in <CONST>, rather than with the value “0,” as Google states. In other words, rather than the “exit_d” routine having the code `[rPC] = 0`, the routine would have the code `[rPC] = <CONST>`. After all, according to Google, the point of storing the original instruction is to preserve it for later execution. However, the *absence* of such code `[rPC] = <CONST>` is most telling as evidence that Magnusson does not disclose <CONST> storing an original instruction and does not disclose the TRANSLATED instruction overwriting an original instruction. A843-A844 (¶¶27-29); A1031-A1033 (¶¶14-16).

Accordingly, neither the “exit_d” routine nor Magnusson’s description of this routine provides any express or inherent disclosure of “overwriting,” as recited in claims 2-4, 15, 16, and 18-21. Therefore, Magnusson’s “exit_d” routine does not disclose the “overwriting” step of the ’205 Patent claims at issue.

F. "Overwriting" Was Found Only In Google's Expert Testimony

As described herein, Magnusson does not disclose "overwriting," because (a) Magnusson's "introduc[ing]" and Figure 4 neither expressly disclose nor necessarily require "overwriting," as properly construed; (b) Magnusson's "exit_0" routine does not disclose "overwriting," but rather describes loading an instruction description and then branching to a service routine; and (c) Magnusson's "exit_d" routine does not disclose "overwriting," rather it describes replacing the current instruction with "0" so that the current instruction is not executed, and then branching to an exit routine. Magnusson, in short, is deficient, and Google improperly used expert testimony to support its anticipation position whether express or inherent.

1. For express anticipation, expert testimony cannot supply missing gaps in Magnusson

If one considers Google's theory of anticipation to be express, Google went beyond Magnusson's deficient disclosure and relied heavily on extrinsic evidence, *i.e.*, Google's expert declarations, in an attempt to supply elements for the gaps of the anticipation rejection.

For example, Google cited expert testimony of Dr. Conte for the purported disclosure of the "overwriting" step by Magnusson as follows: "Specifically, Figure 4 depicts the TRANSLATED instruction after it has been 'introduced' or written into the interpreted program. (See Conte Decl. at ¶¶ 19, 26.)." A1230. As

discussed *supra*, Figure 4 has no such illustration and, thus, Google was seeking to buttress Magnusson's deficient disclosure by citing expert testimony.

Google further cited to expert testimony for the purported disclosure of the "overwriting" step in the "exit_0" routine, as follows: "In other words, since Magnusson is discussing the replacement of an original instruction with the new TRANSLATED instruction, Magnusson also discloses the storage of the original virtual machine instruction into the <CONST> constant value. (See Conte Decl. at ¶¶ 29-30.)." A1230. Again, as discussed *supra*, Magnusson does not disclose <CONST> storing an original instruction and does not disclose the TRANSLATED instruction overwriting an original instruction and, thus, Google was again seeking to buttress Magnusson's deficient disclosure by citing expert testimony.

Google also cited to expert testimony for the purported disclosure of the "overwriting" step in the "exit_d" routine, as follows: "It does this by calling the [rPC] = 0 instruction, which shows that the virtual machine instruction in the intermediate code is over-written with the value '0.' (See Conte Decl. at ¶ 33.)." A1231. As discussed *supra*, the value "0" is not an instruction or virtual machine instruction and therefore cannot be executed and, thus, Google was yet again seeking to buttress Magnusson's deficient disclosure by citing expert testimony.

Such reliance on an expert's testimony to buttress a deficient reference is improper. Quite simply, express anticipation requires that the *reference* disclose each and every element of a claim. *See, e.g.*, *Finnigan*, 180 F.3d at 1365. The fact that Google had to repeatedly go beyond Magnusson's four corners and instead rely on extrinsic evidence provides a clear indication that Magnusson does not disclose "overwriting." There are certainly times when extrinsic evidence can be used to understand a prior art reference in the context of anticipation, but it is error to *supply* a missing limitation in a deficient prior art reference with extrinsic evidence. *See Studiengesellschaft Kohle, m.b.H. v. Dart Indus., Inc.*, 726 F.2d 724, 727 (Fed. Cir. 1984).

2. For inherent anticipation, expert testimony on how Magnusson possibly operates is insufficient

It appears, however, that Google is not relying on express anticipation, but rather inherent anticipation. Indeed, Google's reliance on the understanding of one of skill in the art to prove anticipation means that Google is relying on an inherency theory for anticipation. *Finnigan*, 180 F.3d at 1365.

Even under inherent anticipation, Google has improperly used expert testimony. Dr. Conte's testimony does not cite any other corroborating evidence regarding "overwriting." It is therefore conclusory and unhelpful when determining what Magnusson *must* have disclosed for the purposes of inherent anticipation and, at best, provides merely speculation and possibilities which is

Case: 14-1351 Document: 20 Page: 57 Filed: 05/13/2014

insufficient for inherent anticipation. For example, Dr. Conte merely concludes that, in the "exit_0" routine, an original virtual machine instruction is stored in <CONST> and the TRANSLATED instruction is inserted in place of the original instruction based on a statement in Magnusson: "<CONST> is some constant known at compile time." A1230-A1231 (quoting A489). Yet Dr. Conte provides no support for concluding that "some constant" is an original virtual machine instruction that is overwritten by the TRANSLATED instruction. Dr. Conte provides no helpful explanation about how a person of ordinary skill in the art would understand such is inherently disclosed based on that short statement in Magnusson.

In another example, Dr. Conte merely concludes that, in the "exit_d" routine, the TRANSLATED instruction is overwritten with "0" and therefore must itself be capable of overwriting an original instruction based on a code instruction "[rPC] = 0" in Magnusson (A489). A1231. Yet, the testimony provides no support for concluding that rPC contains or is even related to the TRANSLATED instruction so as to be overwritten by "0." Moreover, Dr. Conte provides no support that, even if the code instruction demonstrated that the TRANSLATED instruction could be overwritten, there is no indication that the TRANSLATED instruction itself performs overwriting in the "exit_d" routine. Again, Dr. Conte

provides no helpful explanation about how the skilled person would understand such is inherently disclosed based on that code instruction in Magnusson.

These are only two of many examples showing how Google's expert testimony is conclusory and unhelpful in determining what Magnusson may or may not inherently disclose.

In short, the Examiner and Google's reliance on Google's expert testimony to expand the scope and content of Magnusson is improper in the context of an express anticipation rejection. And, even when considered, the testimony fails to bridge these gaps required for inherent anticipation. At best, the testimony provides speculation and possibilities, but that is clearly insufficient under established case law. Accordingly, the anticipation rejection of claims 2-4, 15, 16, and 18-21 affirmed by the Board should be reversed.

III. MAGNUSSON IS NOT ENABLED

Magnusson certainly does not disclose the claimed "overwriting," but more fundamentally, it is not enabled. Its lack of enablement directs the reversal of not only the anticipation rejection of "overwriting" claims 2-4, 15, 16, and 18-21, but also the anticipation rejection of claims 1 and 8. As a result, during the reexamination, Oracle set forth a detailed record of why Magnusson is not enabled. *See, e.g.*, A840-A844 (¶¶15-29); A1028-A1031 (¶¶4-13). But the Board—in just over a page—concluded that Magnusson was enabled. A10-A11.

The Board erred. First, Magnusson does not provide an enabling disclosure on the key issue of how a TRANSLATED instruction is to be introduced, and certainly fails to describe how it is to be introduced in a manner meeting the limitations of the claims at issue. Second, the Board largely followed the Examiner's and Google's improper comparison of the amount of disclosure in the '205 Patent with the amount of disclosure in Magnusson to determine Magnusson's enablement. A11; A26; A1227-A1228. Third, the Examiner erroneously applied a higher level of skill in the art than mandated in the law to determine Magnusson's enablement, a point not discussed by the Board. A28. For at least these reasons, the Examiner and Google's enablement analyses are fatally flawed and should have been disregarded by the Board. As Oracle has previously presented, Magnusson is not enabling with respect to the claimed features of the '205 Patent and therefore cannot anticipate the at issue claims of the '205 Patent. A1186-A1189.

A. Magnusson Does Not Describe How Its TRANSLATED Instruction Is Introduced So As To Be Enabling

During the reexamination, the Examiner and Google relied on code examples and Figure 4 of Magnusson to assert that Magnusson is enabled. Oracle repeatedly explained that the code examples were inadequate and had errors, particularly with respect to the introduction of the TRANSLATED instruction, which is critical to Google's anticipation arguments. The Examiner, Google—and

now the Board—erred in dismissing or not addressing these identified inadequacies and discrepancies in reaching a conclusion of enablement. A10-A12; A33-A35.

1. Magnusson's code examples are inadequate

Magnusson's code examples are not adequate to enable a person skilled in the art to practice the claimed invention, because the code examples lack a *key* component in the Examiner and Google's assertions that Magnusson anticipates the '205 Patent claims at issue. Specifically, the examples lack any code showing the introduction of the TRANSLATED instruction, which the Examiner and Google assert meets the "generating" step and the "representing" step of exemplary claims 1 and 8. As such, the lack of an example showing this key feature of the purported system further supports Oracle's position that Magnusson was not enabled as to the claimed features of the '205 Patent claims at issue. A841-A843 (¶¶20-25); A1028-A1030 (¶¶4-9).

First, the "prototype implementation" code in Magnusson (at A491-A492) merely shows target M88100 code and its corresponding SPARC code. *Nowhere does this example show the introduction of the TRANSLATED instruction and how to execute the SPARC code using the TRANSLATED instruction.* This code is therefore inadequate for enabling Magnusson's disclosure with respect to the '205 Patent claims at issue. A1028 (¶4).

Second, Google's cited portion of the hand-coded "partial translation" example in Magnusson is similarly lacking. A488. The cited portion shows translated pseudo-code. *Nowhere does this example show the introduction of a TRANSLATED instruction and how the TRANSLATED instruction is used to execute the pseudo-code*. A1028-A1029 (¶¶5-6).

Had Magnusson's disclosure of the TRANSLATED instruction been "well-documented and well-explained" as Google concluded (A1048), Magnusson would have provided a code example for introducing the TRANSLATED instruction. A487. However, Dr. Goldberg testified that because the examples in Magnusson lack any code showing the introduction of the TRANSLATED instruction, the code lacks sufficient details so as to enable a person of ordinary skill to introduce a TRANSLATED instruction, without undue experimentation, and therefore cannot anticipate the '205 Patent claims at issue. A841-A843 (¶¶20-25); A1030 (¶8).

2. Magnusson has errors

The Examiner erred in dismissing the significance of the discrepancies in Magnusson's description and code examples, and the Board erred in not addressing the discrepancies. A10-A12; A33-A35. Indeed, the overall lack of detail in Magnusson about the key component of the anticipation rejection—the introduction of the TRANSLATED instruction—makes it even more imperative

that what little description Magnusson provided is accurate. Disturbingly, what little description is provided by Magnusson is clearly erroneous.

First, there is a discrepancy between the "partial translation" code and Magnusson's disclosure. The following step in the "partial translation" example reads:

```
rPC = rPC + 24                    ; point to next instruction
```

A488. The discrepancy lies in the use of "24." The code states that the instruction pointer "rPC" is incremented by 24 to point to the next instruction following the block. However, Magnusson clearly states that "[t]he intermediate code is in a 64-bit format." A486. As such, the code as written in Magnusson would not point to the next instruction following the block because 24 bytes would only advance the PC register by three instructions within the block, not the six needed to execute the next instruction following the block correctly. Therefore, according to Dr. Goldberg, it is unclear (a) whether Magnusson meant to point to the next instruction following the block and "24" is a typographical error, or (b) whether Magnusson uses a different instruction size in this example such that "24" does move the pointer to the next instruction following the block. A1028-A1029 (¶¶5-6).

The confusion caused by this discrepancy carries over to the "exit" block in the "partial translation" example which relies on the "rPC" value, as follows:

```
rOP = [rPC]        ; get next instruction specification
r1 = rOP & rMASK   ; get the service routine offset
. . .
br rCODE[r1]       ; otherwise dispatch next instruct.
```

A489. Here Magnusson describes "rOP" as being set to the next instruction specification pointed to by "rPC." Magnusson then describes setting "r1" to point to a service routine offset by masking certain bits of the instruction. Magnusson then describes branching to the service routine found at the offset contained in "r1." However, given that Magnusson states that "[t]he intermediate code is in a 64-bit format," this code would not work because the value of "[rPC]," loaded into "rOP," could not contain both the address of the service routine and the parameters required in "rOP." As with the previous example, it may be the case that Magnusson intended for the intermediate format instruction size to be different than 64 bits, or the code is simply erroneous. A1029-A1030 (¶7).

These examples of unresolvable discrepancies, combined with no disclosure of how the TRANLSATED instruction is introduced, illustrate that Magnusson is not enabling with respect to the at issue claims of the '205 Patent.

B. Enablement Analysis Should Be Based On Magnusson Alone

1. The Board's analysis was inconsistent with *Morsa*

The Board erred in deferring to the Examiner's and Google's comparison of the level of detail in the '205 Patent with the level of detail in Magnusson to establish that Magnusson is enabling. A11; A26; A1227-A1228. The Board's

position is inconsistent with *In re Morsa*, 713 F.3d 104 (Fed. Cir. 2013). *Morsa* squarely held that "an examiner must determine if prior art is enabling . . . based on the disclosure *of that particular document*," *i.e.*, the prior art reference and not the disclosure of the patent. *Id.* at 110 (emphasis in original). The Board largely dismissed this error, demonstrating a lack of understanding of the thrust and caution provided by *Morsa*. A11.

In *Morsa*, the examiner rejected the applicant's claims as unpatentable over a prior art publication. *In re Morsa*, 713 F.3d at 107. The applicant in *Morsa* argued that the publication was not enabling because it lacked specific disclosures of certain elements, how the elements were integrated, and the corresponding process. *Id.* at 108. On appeal to this Court, the Patent Office argued that the publication was enabling because it was "at least as enabling" as the applicant's application. *Id.* at 110. This Court disagreed. It found that the consideration of the application was relevant for determining the scope of the claimed invention, but not for whether a prior art reference is enabling. *Id.* Indeed, according to the *Morsa* Court, "an examiner must determine if prior art is enabling by asking whether a person of ordinary skill in the art could make or use the claimed invention without undue experimentation based on the disclosure *of that particular document*." *Id.* (emphasis in original).

Case: 14-1351 Document: 20 Page: 65 Filed: 05/13/2014

Like *Morsa*, the Examiner and Google here have attempted to find Magnusson enabling by comparing its disclosure to that of the ’205 Patent. Google argued that “Magnusson in fact provides much more support than does the ’205 Patent” to justify its conclusion that Magnusson is enabling. A1226. Similarly, the Examiner argued that “the third party requester’s [Google’s] observation that the patent owner did not provide the type of detail in the ’205 patent specification that the patent owner now argues is necessary in Magnusson supports a finding that one skilled in the art would have known how to implement the features of Magnusson and would have concluded that the reference disclosures would have been enabling.” A26.

The Board dismissed this error, and demonstrated a lack of understanding of the thrust and caution of *Morsa*, stating: “We note that, despite urging an error in the comparison between the disclosure of the ’205 patent and the disclosure of Magnusson, Owner engages in this comparison to attempt to demonstrate the superior nature of the disclosure within the ’205 patent.” A11. Just like the Patent Office in *Morsa*, the Board improperly analyzed Magnusson’s lack of enablement based on the disclosure of the ’205 Patent and improperly shifted to the disclosure of the ’205 Patent. Accordingly, *Morsa* dictates the reversal of the rejection.

2. Comparison to the ’205 Patent highlights Magnusson’s lack of enablement

Even if a comparative analysis between the ’205 Patent and Magnusson is performed (in contravention of *Morsa*), the analysis supports a finding that Magnusson is not enabled. A comparison of Magnusson’s Figure 4 to the ’205 Patent’s Figure 5 could not be more telling. The ’205 Patent’s Figure 5 shows the “overwriting” of an original virtual machine instruction (“BYTECODE 2”) with a new virtual machine instruction (“GO_NATIVE #N”). A77.

Moreover, the ’205 Patent describes “generating” and “executing” the new GO_NATIVE virtual machine instruction:

> FIG. 5 shows a generation of hybrid virtual and native machine instructions. Java virtual machine instructions 301 are bytecodes where each bytecode may include one or more bytes. The Java virtual machine instructions typically reside in a Java class file as is shown in FIG. 3. In the example shown, the interpreter decides to introduce a snippet for bytecodes 2-5 of virtual machine instructions 301. The interpreter *generates* modified Java virtual machine instructions 303 by overwriting bytecode 2 with a go_native virtual machine instruction….
>
> . . . .
>
> When the interpreter *executes* the go_native bytecode, the interpreter will look up the snippet in the snippet zone specified by the go_native bytecode and then activate the native machine instructions in the snippet.

A89 (7:63-8:5, 8:27-30) (emphasis added).

In contrast, Magnusson's Figure 4 only shows a "*translated*" simulation code with no indication overwriting anything. A486. Moreover, Magnusson merely describes "we introduce a new instruction, TRANSLATED," with no indication of how the instruction is generated or executed:

> [W]e *introduce* a new instruction, TRANSLATED. This instruction takes as a parameter a pointer to the translated block, and handles generic entry/exit issues.

A487 (emphasis added). Given the stark differences between the two disclosures, the Board's finding of enablement—even based on the '205 Patent—is incorrect. Hence, the Board's affirmance of the Examiner's rejection of claims 1-4, 8, 15, 16, and 18-21 based on Magnusson should be reversed for this reason.

C. The Board's Enablement Analysis Misapplied The Required Level of Skill

The Board has further erred in misapplying the requisite level of skill in the enablement analysis by deferring to the Examiner. A11; A28. Oracle relied on its expert to identify inadequacies and discrepancies in Magnusson. This analysis was to show how a person of ordinary skill following Magnusson would be led to a dead end. The Examiner took this expert analysis and twisted it to support his enablement argument. In particular, the Examiner asserted that "the patent owner's own argument [citing to Patent Owner's expert declaration] appears to demonstrate that the Figure and accompanying text would have been readily understandable despite the noted inconsistency." A28. In other

words, the Examiner argued that, because Oracle's expert was able to point out the discrepancies and errors in Magnusson, Magnusson must be "readily understandable" and therefore enabling to a person of ordinary skill in the art. A28.

The Examiner is incorrect at least because Oracle's expert in the reexamination is a person of *extraordinary* skill in the art, not a person of *ordinary* skill in the art, as required by 35 U.S.C. § 112 (pre-AIA). *See, e.g.*, *Environmental Designs, Ltd. v. Union Oil Co. of Cal.*, 713 F.2d 693, 697 (Fed. Cir. 1983) ("a person of 'ordinary skill in the art'—not . . . geniuses in the art at hand"). The fact that Oracle's expert was able to identify Magnusson's errors and discrepancies does not mean at all that the person of ordinary skill would be able to identify them as well (or that the one of ordinary skill in the art would know how to correct the errors and discrepancies). Like an incorrect map, Magnusson would have led astray the person of ordinary skill without that person necessarily knowing the errors.

Indeed, the fact that it took an *expert* to point out the discrepancies and errors in Magnusson *confirms* Magnusson's lack of enablement. That is, the need for an expert, many years removed, to understand Magnusson sufficiently so as to recognize and correct any errors and discrepancies therein at least calls into doubt that the person of ordinary skill could have done so. Rather, the person of

ordinary skill would have been left with an erroneous, inconsistent disclosure, thereby defeating enablement.

Moreover, the fact that Google's expert, also a person of *extraordinary* skill, was necessary to explain what Magnusson is purported to disclose is further evidence that Magnusson lacks enablement. A1228-A1232. Hence, the Examiner's enablement analysis, and the Board's reliance thereon, is flawed and should be disregarded. Magnusson is not enabling with respect to the at issue claims of the '205 Patent.

CONCLUSION

As presented, the Board erred in construing "overwriting" in claims 2-4, 15, 16, and 18-21 as an act of replacing some information in a computer file with new information, rather than literally writing over an existing information. The plain claim language, the specification and other claims support a construction of "overwriting" as writing over. This Court should accordingly remand to conduct a new invalidity analysis using the proper construction of "overwriting."

If the Court wishes to address whether Magnusson anticipates claims 2-4, 15, 16, and 18-21 under the proper construction of "overwriting," the record demonstrates that Magnusson does not expressly or inherently disclose any "overwriting." The Court should accordingly reverse the Board's decision to

affirm the Examiner's rejection of claims 2-4, 15, 16, and 18-21 as anticipated by Magnusson.

Finally, the Board erred in affirming the Examiner's finding that Magnusson is an enabling reference so as to anticipate claims 1-4, 8, 15, 16, and 18-21. Accordingly, the Court should reverse the Board's decision to affirm the rejection of claims 1-4, 8, 15, 16, and 18-21 of the '205 Patent.

Respectfully submitted,

MAY 13, 2014

/s/ Mehran Arjomand

MEHRAN ARJOMAND
MORRISON & FOERSTER LLP
707 Wilshire Boulevard
Los Angeles, CA 90017
Telephone: (213) 892-5630
Facsimile: (213) 892-5454
Counsel for Appellant Oracle America, Inc.

ADDENDUM

Case: 14-1351 Document: 20 Page: 72 Filed: 05/13/2014

ADDENDUM

TABLE OF CONTENTS

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
95/001,548	02/17/2011	6910205	13557.112021	1709

25226 7590 11/27/2013
MORRISON & FOERSTER LLP
755 PAGE MILL RD
PALO ALTO, CA 94304-1018

EXAMINER	
KISS, ERIC B	
ART UNIT	PAPER NUMBER
3992	
MAIL DATE	DELIVERY MODE
11/27/2013	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

PTOL-90A (Rev. 04/07)

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

GOOGLE, INC.
Requester and Respondent

v.

ORACLE AMERICA, INC.
Patent Owner and Appellant

Appeal 2013-010321
Reexamination Control 95/001,548
Patent US 6,910,205 B2
Technology Center 3900

Before STEPHEN C. SIU, DENISE M. POTHIER, and ANDREW J. DILLON, *Administrative Patent Judges*.

DILLON, *Administrative Patent Judge*.

DECISION ON APPEAL

STATEMENT OF THE CASE

Owner appeals under 35 U.S.C. § 134(b) (2002) from the final decision of the Examiner adverse to the patentability of claims 1-4, 8, 15, 16, and 18-21. An oral hearing was conducted on November 13, 2013. We have jurisdiction under 35 U.S.C. § 315 (2002).

We affirm.

Invention

United States Patent No. 6,910, 205 ("'205 patent") describes a method for increasing the execution speed of virtual machine instructions for a function. A portion of the virtual machine instructions for the function are compiled into native machine instructions so that the function includes both virtual and native machine instructions. Execution of the native machine instructions may be accomplished by overwriting a virtual machine instruction of the function with a virtual machine instruction that specifies execution of the native machine instructions. '205 patent, Abstract.

Figure 5 of the '205 patent is depicted below:

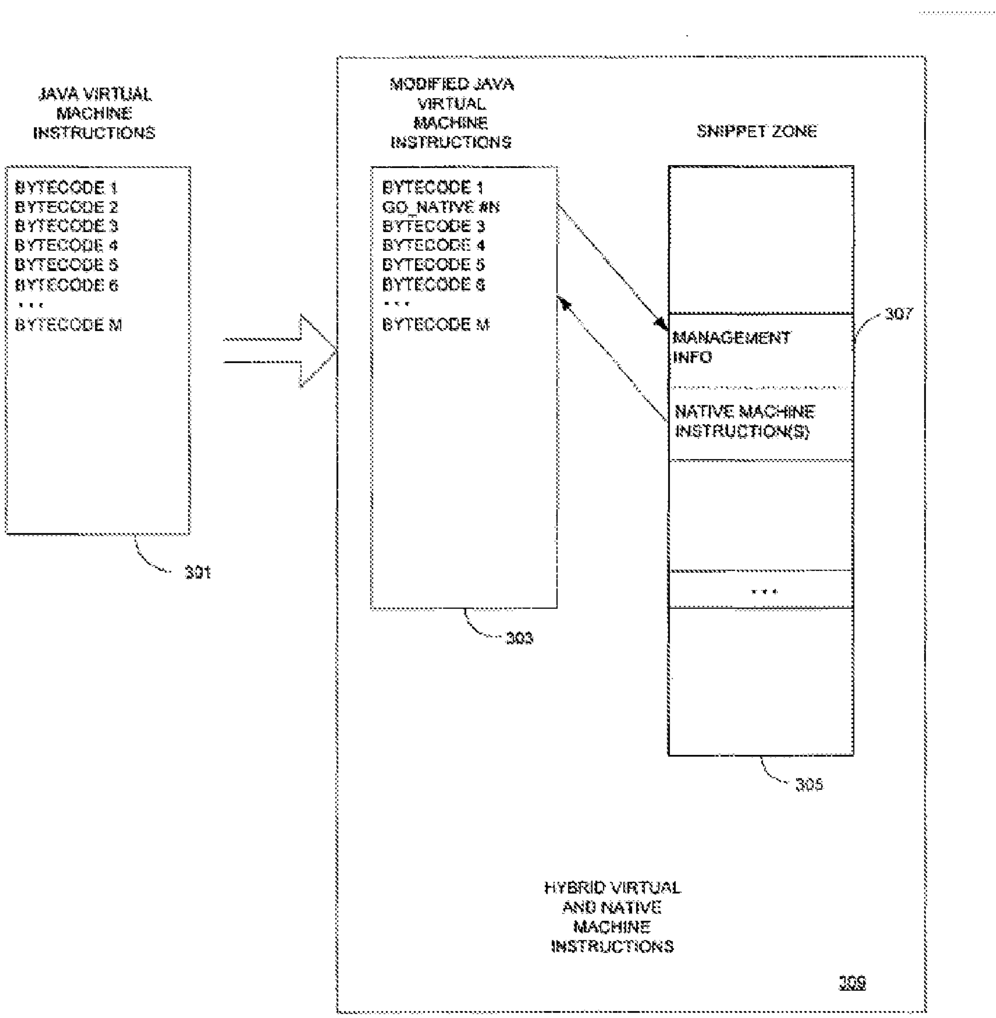


Figure 5 depicts a transformation of Java virtual machine instructions of a function to hybrid virtual and native machine instructions. '205 patent, col. 3, ll. 46-48.

Java virtual machine instructions typically reside in a Java class file, as illustrated at reference numeral 301. In the example depicted, the interpreter

introduces a snippet for bytecodes 2-5 of virtual machine instructions 301. As illustrated at reference numeral 303, this is accomplished by overwriting bytecode 2 with a go_native virtual machine instruction. The go_native virtual machine instruction points to a snippet 307 within snippet zone 305. As described, each snippet includes two sections. A management section, which stores the original bytecode 2 and the original address of bytecode 2; and, a second section which includes a sequence of one or more native machine instructions. The native machine instructions within the snippet perform the same operations as if bytecodes 2-5 had been interpreted. Afterwards, the interpreter continues with the execution of bytecode 6, as if no snippet existed. '205 patent, col. 7, l. 63 – col. 8, l. 34.

Claims

Claims 1-4, 8, 15, 16, and 18-21 are subject to reexamination and have been rejected. Claims 1-14 are original patent claims. Claim 17 has been canceled. Claims 15, 16, and 18-21 are proposed new claims. Claims 1, 8, 15, 16, 18, and 21 are independent.

Claim 8 is illustrative.

8. In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:

inputting virtual machine instructions for a function;

compiling, at runtime, a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instruction;

representing said at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the function.

Prior Art

Peter Magnusson	*Partial Translation,* Swedish Institute of Computer Science Technical Report (T93.5), October 1993 (hereinafter "Magnusson")

Owner's Contentions

Owner contends that the Examiner erred in rejecting claims 1-4, 8, 15, 16, and 18-21 as anticipated by Magnusson under 35 U.S.C. §102(b). App. Br. 11.

ANALYSIS

Owner argues that Magnusson does not anticipate the appealed claims of the '205 patent for the following reasons:

A. Magnusson does not disclose the "overwriting" step of claims 2, 15, 16, 18, and 21.
B. Magnusson does not disclose the "generating" and "executing" steps of claims 1, 15, and 16 when the phrase "virtual machine instruction" is properly interpreted.
C. Magnusson does not disclose the "representing" step of claims 8, 18, and 21.
D. Magnusson is not enabled so as to anticipate the appealed claims of the '205 patent.

Claim Interpretation

In this proceeding, the claim language should be read in light of the specification as it would be interpreted by one of ordinary skill in the art. *In re Am. Acad. of Sci. Tech. Ctr.*, 367 F.3d 1359, 1364 (Fed. Cir. 2004). The Office must apply the broadest reasonable meaning to the claim language, taking into account any definitions presented in the specification. *Id.* (citing *In re Bass*, 314 F.3d 575, 577 (Fed. Cir. 2002)).

There is also a "heavy presumption" that a claim term carries its ordinary and customary meaning. *CCS Fitness, Inc. v. Brunswick Corp.*, 288 F.3d 1359, 1366 (Fed. Cir. 2002).

"Overwriting"

Owner contends that Magnusson fails to disclose "overwriting" (App. Br. 11-12); however, Owner does not point to any disclosure in the '205 patent that sets forth a clear definition for "overwriting."

Consequently, we must look to the disclosure of the '205 patent to determine a proper definition for "overwriting." At column 8, lines 1-5, the '205 patent describes an example wherein the interpreter decides to introduce a snippet for bytecodes 2-5 of virtual machine instructions 301. The specification then describes "overwriting bytecode 2 with a go_native virtual machine instruction." The '205 patent then states that the original bytecode 2 and the original address thereof are stored within a section of management information within snippet 307. Snippet 307 also includes one or more native machine instructions which are

utilized to replace bytecodes 2-5. Thereafter, the process continues with the execution of bytecode 6, "as if no snippet existed." '205 patent, col. 8. ll. 11-34.

We note that the explicit illustration within the '205 patent is that only bytecode 2 has been "overwritten" within modified Java virtual machine instructions 303, despite the fact that bytecodes 3-5 are also being replaced by one or more native machine instructions. *See* '205 patent, Fig. 5. We therefore interpret the term "overwriting" as the act of replacing some information in a computer file with new information, rather than literally writing over an existing information.

Magnusson

Owner argues that Magnusson does not anticipate the rejected claims of the '205 patent because, *inter alia,* Magnusson does not disclose the "overwriting" step of claims 2, 15, 16, 18 and 21. App. Br. 12.

Specifically, Owner alleges that the Examiner's reliance on the teaching within Magnusson of "introduc[ing]" a TRANSLATED instruction as disclosing "overwriting" is incorrect. Owner argues that "introducing" neither explicitly nor inherently requires "overwriting." Owner further argues that Magnusson does not utilize the term "overwriting," that "nothing is shown on how the 'translated' instruction came to be in the simulation code" and that "[t]he key step that performs 'overwriting' is simply not there." *Id* at 15.

Requester submits that Figure 4 of Magnusson illustrates the introduction of a new instruction into the intermediate code and consequently the new instruction must "overwrite" existing code. The Examiner concurs with Requester. RAN 30, 38-39 (citing Magnuson 8-9).

We find that Magnusson discloses the use of intermediate code instructions for an emulated processor and then jumping to a native code (SPARC) block upon encountering a TRANSLATED instruction. Under the broad but reasonable interpretation of "overwriting" previously forth, we find this process replaces instructions within the intermediate code with new information and therefore find that Magnusson discloses "overwriting."

Owner argues that the Examiner erred in finding that Magnusson discloses the "generating" and "executing" steps of claims 1, 15, and 16. In particular, Owner argues that Magnusson fails to disclose "generating, at runtime, a new virtual machine instruction" and "executing said new virtual machine instruction instead of said first virtual machine instruction." Owner urges that, given that the Examiner has interpreted the intermediate code of Magnusson as "virtual machine instruction," Magnusson fails as a reference to disclose the "generating" and "executing" steps. In Owner's view, this is because Magnusson does not generate new target code to be executed instead of the other target code, as required by the "generating" step. App. Br. 19-20.

Requester submits that Magnusson discloses code written for a target (e.g., M88100 processor) and then translating that code into an intermediate code. In the Magnuson example, the target code is in a 32-bit format while the intermediate code is in a 64-bit format. Requester urges that both target code and intermediate code are "virtual machine instructions" intended for execution within the software emulated microprocessor and that native code running on the "host" is the scalable processor architecture (SPARC) code. RAN 32 (citing Magnuson at 8).

Thereafter, Requester argues that Magnusson discloses replacing a virtual machine instruction with the TRANSLATED instruction and executing selected virtual code instructions utilizing native code. *Id.*

The Examiner concurs with Requester for the reasons set forth above. *Id.* at 33

We find that both the intermediate code and target code set forth by Magnusson both constitute "virtual machine instructions" and consequently, we agree with the Examiner's finding that Magnusson discloses "generating, at runtime, a new virtual machine instruction" and "executing said new virtual machine instruction instead of said first virtual machine instruction" in that Magnusson discloses generating, at run time, the TRANSLATED instruction and, upon encountering the TRANSLATED instruction, executing that instruction.

Owner also argues that the Examiner erred with respect to the "representing" step of claims 8, 18, and 21; however, beyond a naked assertion that "[a] similar analysis can be done for the 'representing' step of claim 8" (App. Br. 21, Reb. Br. 8.), we find no cogent argument that persuades us of an error in the Examiner's position.

Magnusson Enablement

Owner argues that Magnusson is not enabling in that Magnusson fails to disclose how its TRANSLATED instruction is introduced in a manner that would permit one of ordinary skill in the art to introduce a TRANSLATED instruction. Similarly, Owner argues a lack of enablement for the "generating" and "representing" steps of claims 1, 15, and 16, the "representing" step of claims 8, 18, and 21, and the "overwriting" step of claims 2, 15, 15, 18, and 21. App. Br. 21-22.

Specifically, Owner asserts that the prototype implementation of Magnusson, at pages 13-14, "merely shows target MM88100 code and its corresponding SPARC code" and fails to show the introduction of the TRANSLATED instruction and how to execute SPARC code. *Id.*

Further, Owner urges that various discrepancies in Magnusson's description and code examples were improperly ignored by the Examiner (*id.* at 23) and, that the Examiner erred in performing a comparison between the disclosure of Magnusson and the disclosure contained within the '205 patent.Reb. Br. 2 (citing *In re Morsa,* 713 F.3d 104 (Fed. Cir. 2013)). .

We note that, despite urging an error in the comparison between the disclosure of the '205 patent and the disclosure of Magnusson, Owner engages in this comparison to attempt to demonstrate the superior nature of the disclosure within the '205 patent.

It is well settled that "a prior art printed publication cited by an examiner is presumptively enabling barring any showing to the contrary by a patent applicant or patentee." *In re Antor Media Corp.*, 689 F3d 1282 (Fed. Cir. 2012). After considering the totality of Owner's enablement challenge, and the submissions of Requester, the Examiner found that the record showed, by a preponderance of evidence, that Magnusson is enabling prior art. RAN 10-21.

We have similarly considered the record and we find that the arguments and evidence submitted by Owner are not sufficient to overcome the presumption of enablement relied upon by the Examiner. Consequently, we find that Magnusson is an enabling reference and the Examiner did not err in relying upon that reference as a basis for rejecting claims 1-4, 8, 15, 16, and 18-21.

Accordingly, Owner has not shown reversible error in any of the rejections that have been advanced by the Examiner. In reviewing the record, we also do not discern that the Examiner was incorrect in applying the prior art to the claims involved in this appeal.

Summary/Conclusion

We sustain the Examiner's rejections of claims 1-4, 8, 15, 16, and 18-21.

DECISION

The Examiner's decision adverse to the patentability of claims 1-4, 8, 15, 16, and 18-21 is affirmed.

Requests for extensions of time in this proceeding are governed by 37 C.F.R. §§ 1.956 and 41.79(e).

AFFIRMED

Patent Owner:

MORRISON & FOERSTER LLP
755 Page Mill Road
Palo Alto, CA 94304-1018

Third Party Requester:

KING & SPALDING LLP
1180 Peachtree Street, NE
Atlanta, GA 30309

US006910205B2

(12) United States Patent
Bak et al.

(10) **Patent No.: US 6,910,205 B2**
(45) **Date of Patent: *Jun. 21, 2005**

(54) **INTERPRETING FUNCTIONS UTILIZING A HYBRID OF VIRTUAL AND NATIVE MACHINE INSTRUCTIONS**

(75) Inventors: **Lars Bak**, Palo Alto, CA (US); **Robert Griesemer**, Menlo Park, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 175 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **10/194,040**

(22) Filed: **Jul. 12, 2002**

(65) **Prior Publication Data**

US 2002/0184399 A1 Dec. 5, 2002

Related U.S. Application Data

(63) Continuation of application No. 08/884,856, filed on Jun. 30, 1997, now Pat. No. 6,513,156.

(51) **Int. Cl.**[7] **G06F 9/45**; G06F 9/455

(52) **U.S. Cl.** **717/151**; 717/159; 717/148; 717/139; 718/1

(58) **Field of Search** 717/151, 159, 717/139, 148; 718/1

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,329,611	A	7/1994	Pechanek et al.	
5,367,685	A	11/1994	Gosling	
5,586,328	A	12/1996	Caron et al.	
5,758,162	A	5/1998	Takayama et al.	
5,768,593	A	6/1998	Walters et al.	
5,845,298	A	12/1998	O'Conner et al.	
5,898,850	A	4/1999	Dickol et al.	
5,905,895	A	5/1999	Halter	
5,925,123	A	7/1999	Tremblay et al.	
5,953,736	A	9/1999	O'Conner et al.	
5,995,754	A *	11/1999	Holzle et al.	717/158
6,038,394	A	3/2000	Layes	

(Continued)

OTHER PUBLICATIONS

Proebsting, Todd A., "Optimizing an ANSI C Interpreter with Superoperators," pp. 322–332, Jan. 1995.

Hsieh, Cheng–Hsueh et al., "Java Bytecode to Native Code Translation: The caffeine prototype and preliminary results," pp. 90–97, Dec. 1996.

Lambright, H. Dan., "Java Bytecode Optimizations," pp. 206–210, Feb. 1997.

Pittan Thomas, "Two–level Hybrid Interpreter/Native Code Execution for combined space time program efficiency," ACM, pp. 150–152, Jun. 1987.

Kaufer, Stephen et al., "Saber–C, An Interpreter–based programming environment for the C language," USENIX, pp. 161–171, Jun. 1988.

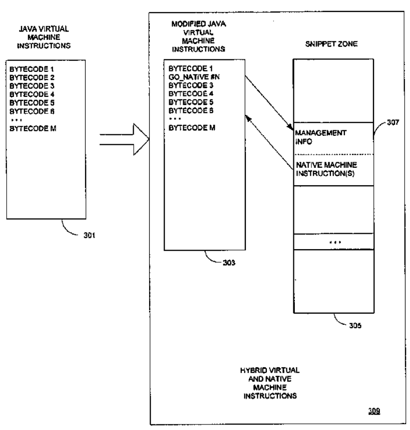
Davidson, Jack W. et al., "Cint: A RISC Interpreter for the C programming language," ACM, pp. 189–198, Jun. 1987.

Primary Examiner—Lewis A. Bullock, Jr.
(74) *Attorney, Agent, or Firm*—Beyer Weaver & Thomas LLP

(57) **ABSTRACT**

Systems and methods for increasing the execution speed of virtual machine instructions for a function are provided. A portion of the virtual machine instructions of the function are compiled into native machine instructions so that the function includes both virtual and native machine instructions. Execution of the native machine instructions may be accomplished by overwriting a virtual machine instruction of the function with a virtual machine instruction that specifies execution of the native machine instructions. Additionally, the original virtual machine instruction may be stored so that the original virtual machine instructions can be regenerated.

14 Claims, 12 Drawing Sheets



U.S. PATENT DOCUMENTS

6,044,220	A		3/2000	Breternitz	
6,118,940	A		9/2000	Alexander et al.	
6,170,083	B1		1/2001	Adl-Tabatabai	
6,292,883	B1	*	9/2001	Augusteijn et al.	712/209
6,332,216	B1		12/2001	Manjunath	
6,349,377	B1	*	2/2002	Lindwer	712/22

* cited by examiner

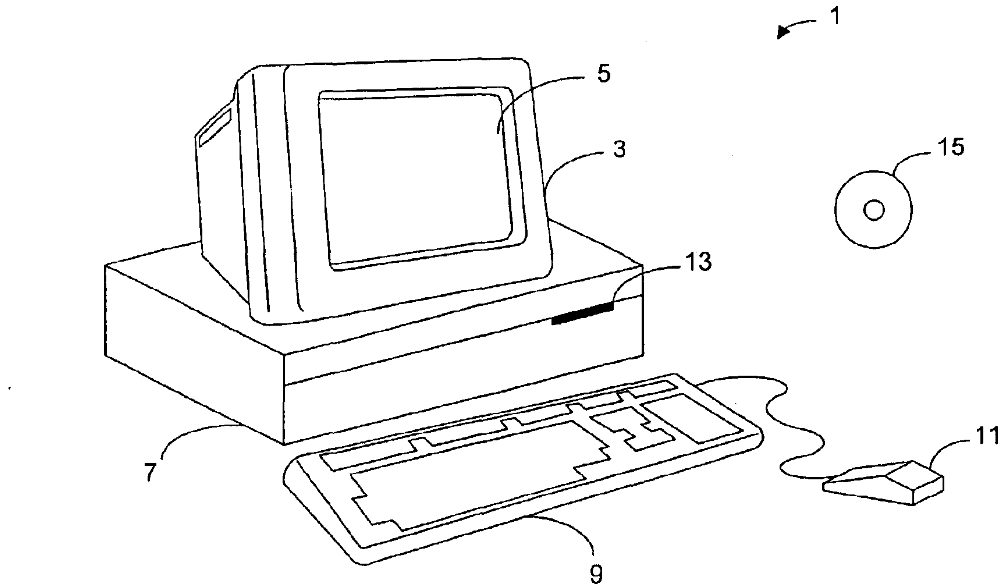

1
5
3
15
13
11
7
9

FIG. 1

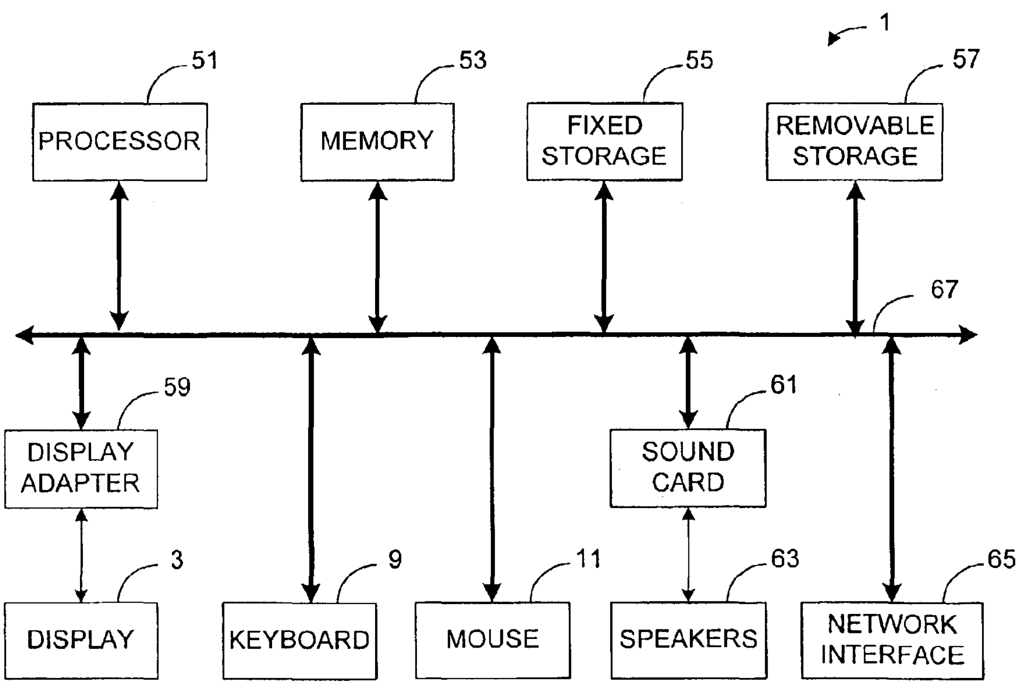

1
51
53
55
57
PROCESSOR
MEMORY
FIXED STORAGE
REMOVABLE STORAGE
67
59
61
DISPLAY ADAPTER
SOUND CARD
3
9
11
63
65
DISPLAY
KEYBOARD
MOUSE
SPEAKERS
NETWORK INTERFACE

FIG. 2

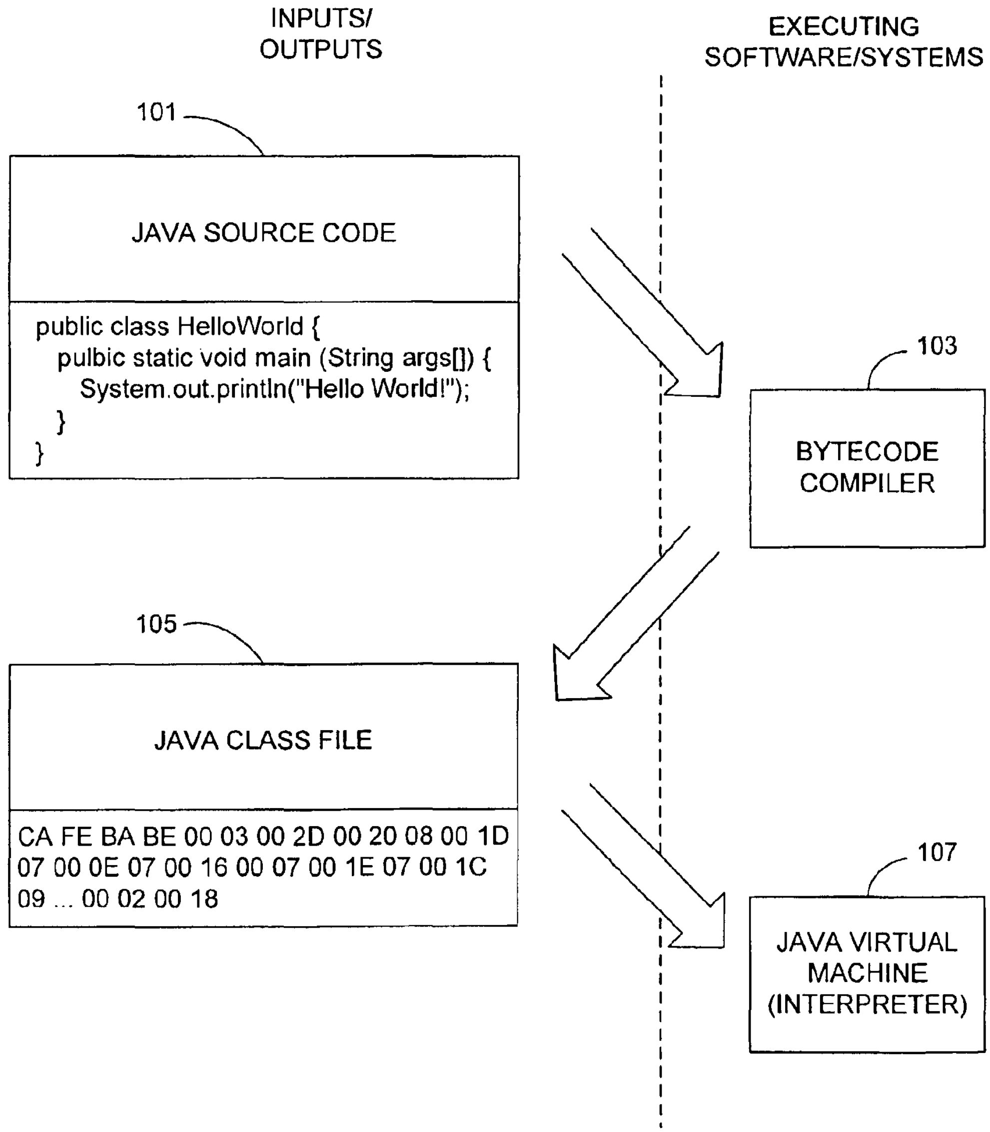

INPUTS/
OUTPUTS
EXECUTING
SOFTWARE/SYSTEMS
101
JAVA SOURCE CODE
public class HelloWorld {
pulbic static void main (String args[]) {
System.out.println("Hello World!");
}
}
103
BYTECODE
COMPILER
105
JAVA CLASS FILE
CA FE BA BE 00 03 00 2D 00 20 08 00 1D
07 00 0E 07 00 16 00 07 00 1E 07 00 1C
09 ... 00 02 00 18
107
JAVA VIRTUAL
MACHINE
(INTERPRETER)

FIG. 3

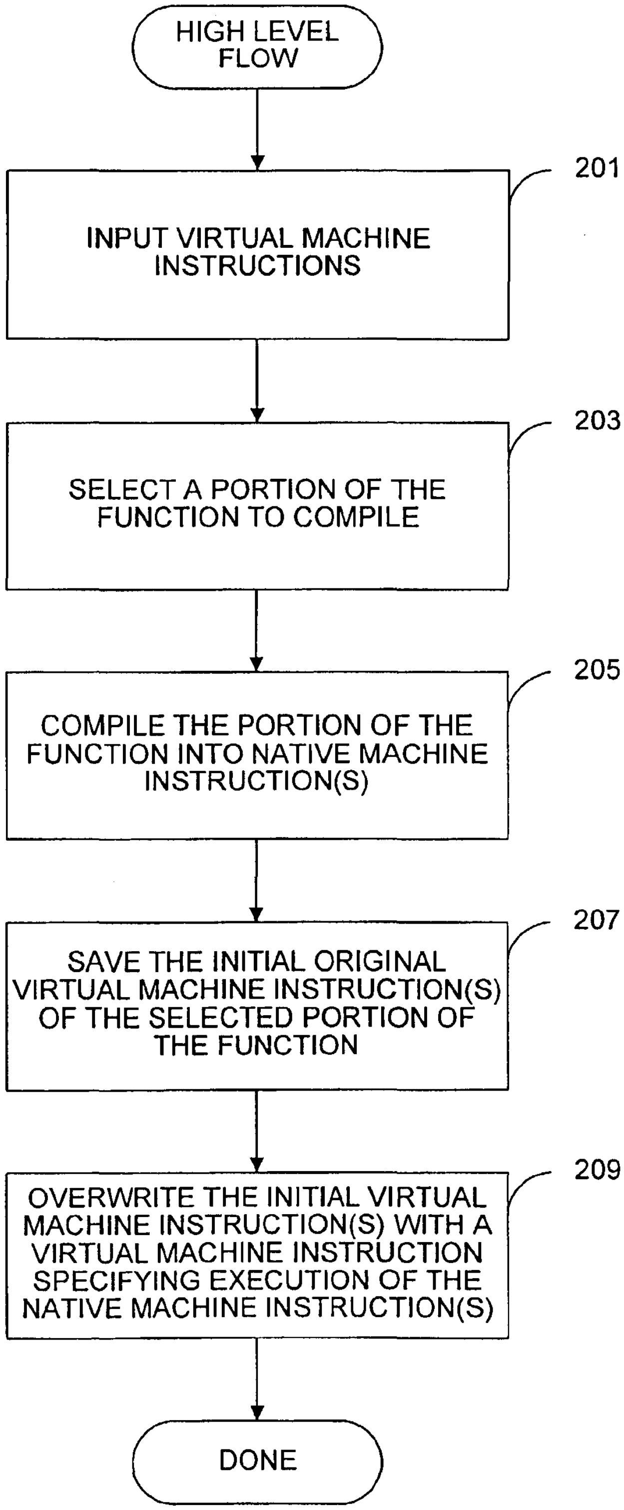

HIGH LEVEL FLOW
201
INPUT VIRTUAL MACHINE INSTRUCTIONS
203
SELECT A PORTION OF THE FUNCTION TO COMPILE
205
COMPILE THE PORTION OF THE FUNCTION INTO NATIVE MACHINE INSTRUCTION(S)
207
SAVE THE INITIAL ORIGINAL VIRTUAL MACHINE INSTRUCTION(S) OF THE SELECTED PORTION OF THE FUNCTION
209
OVERWRITE THE INITIAL VIRTUAL MACHINE INSTRUCTION(S) WITH A VIRTUAL MACHINE INSTRUCTION SPECIFYING EXECUTION OF THE NATIVE MACHINE INSTRUCTION(S)
DONE

FIG. 4

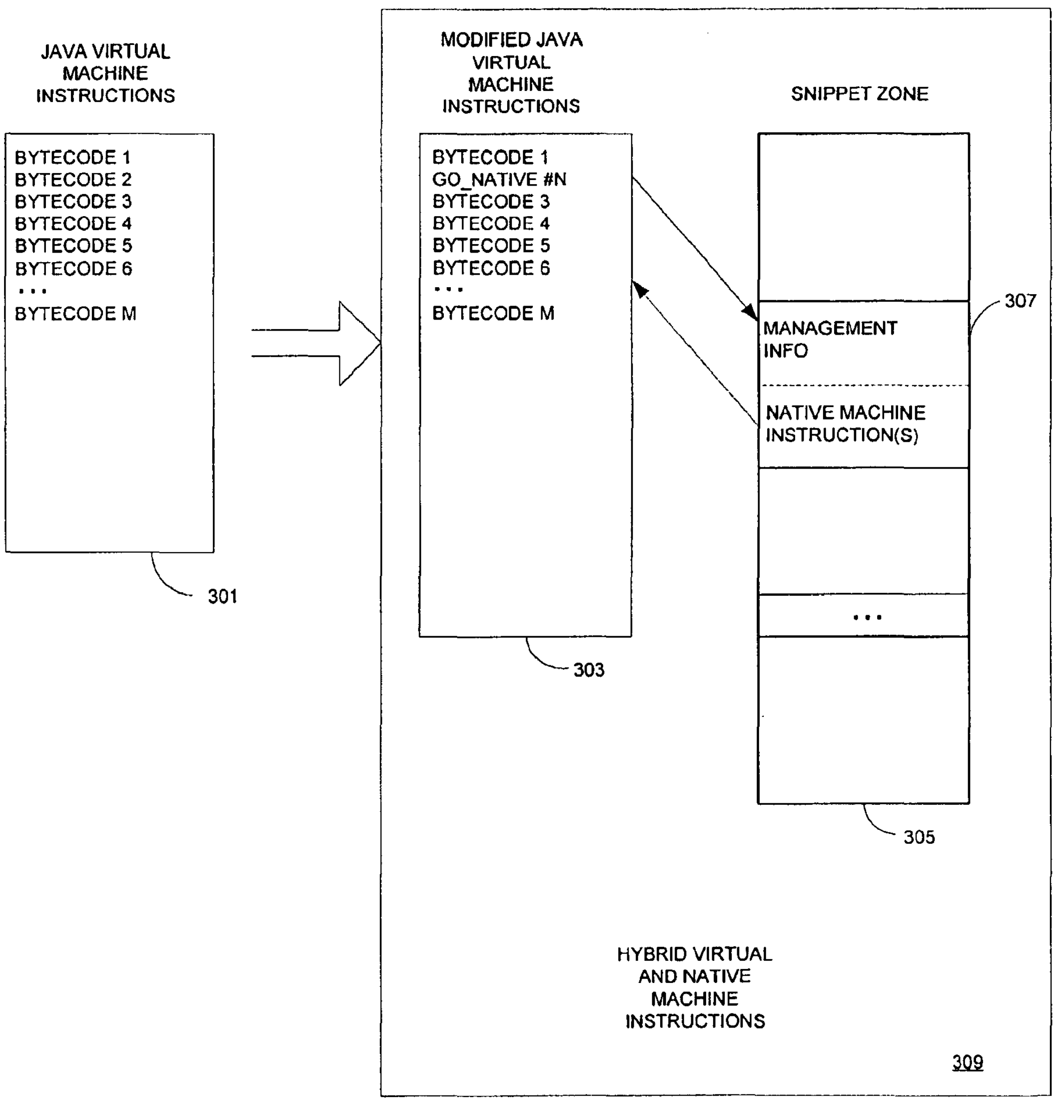

JAVA VIRTUAL MACHINE INSTRUCTIONS
BYTECODE 1
BYTECODE 2
BYTECODE 3
BYTECODE 4
BYTECODE 5
BYTECODE 6
...
BYTECODE M
301
MODIFIED JAVA VIRTUAL MACHINE INSTRUCTIONS
BYTECODE 1
GO_NATIVE #N
BYTECODE 3
BYTECODE 4
BYTECODE 5
BYTECODE 6
...
BYTECODE M
303
SNIPPET ZONE
MANAGEMENT INFO
NATIVE MACHINE INSTRUCTION(S)
307
...
305
HYBRID VIRTUAL AND NATIVE MACHINE INSTRUCTIONS
309

FIG. 5

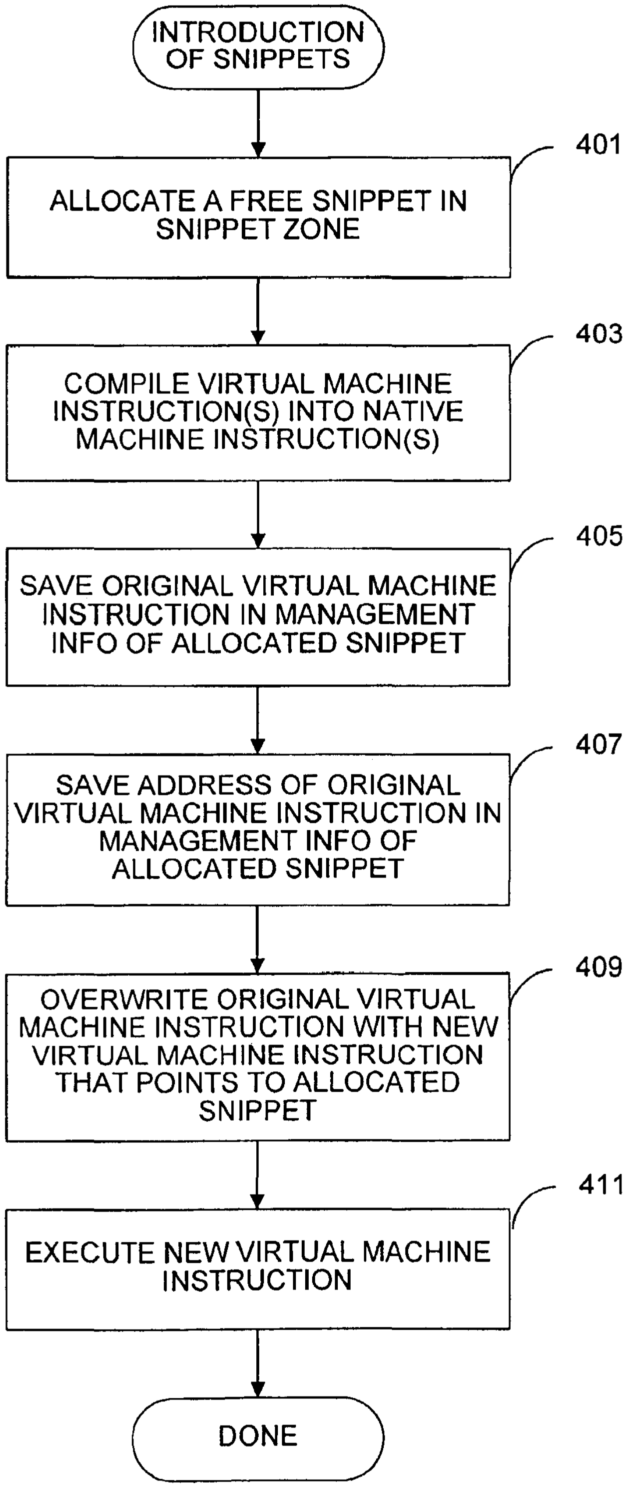

INTRODUCTION OF SNIPPETS
401
ALLOCATE A FREE SNIPPET IN SNIPPET ZONE
403
COMPILE VIRTUAL MACHINE INSTRUCTION(S) INTO NATIVE MACHINE INSTRUCTION(S)
405
SAVE ORIGINAL VIRTUAL MACHINE INSTRUCTION IN MANAGEMENT INFO OF ALLOCATED SNIPPET
407
SAVE ADDRESS OF ORIGINAL VIRTUAL MACHINE INSTRUCTION IN MANAGEMENT INFO OF ALLOCATED SNIPPET
409
OVERWRITE ORIGINAL VIRTUAL MACHINE INSTRUCTION WITH NEW VIRTUAL MACHINE INSTRUCTION THAT POINTS TO ALLOCATED SNIPPET
411
EXECUTE NEW VIRTUAL MACHINE INSTRUCTION
DONE

FIG. 6

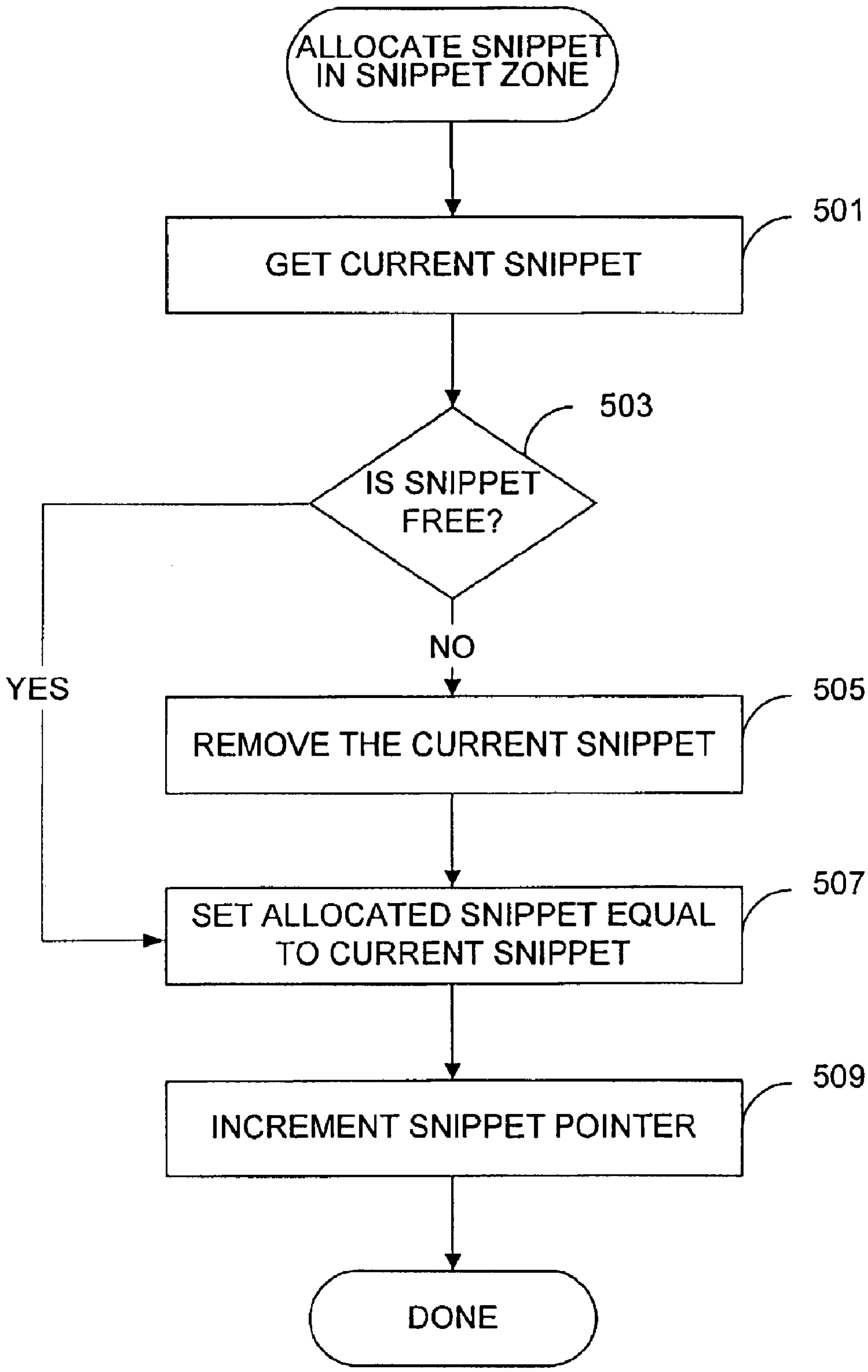

ALLOCATE SNIPPET IN SNIPPET ZONE
GET CURRENT SNIPPET
501
IS SNIPPET FREE?
503
YES
NO
REMOVE THE CURRENT SNIPPET
505
SET ALLOCATED SNIPPET EQUAL TO CURRENT SNIPPET
507
INCREMENT SNIPPET POINTER
509
DONE

FIG. 7

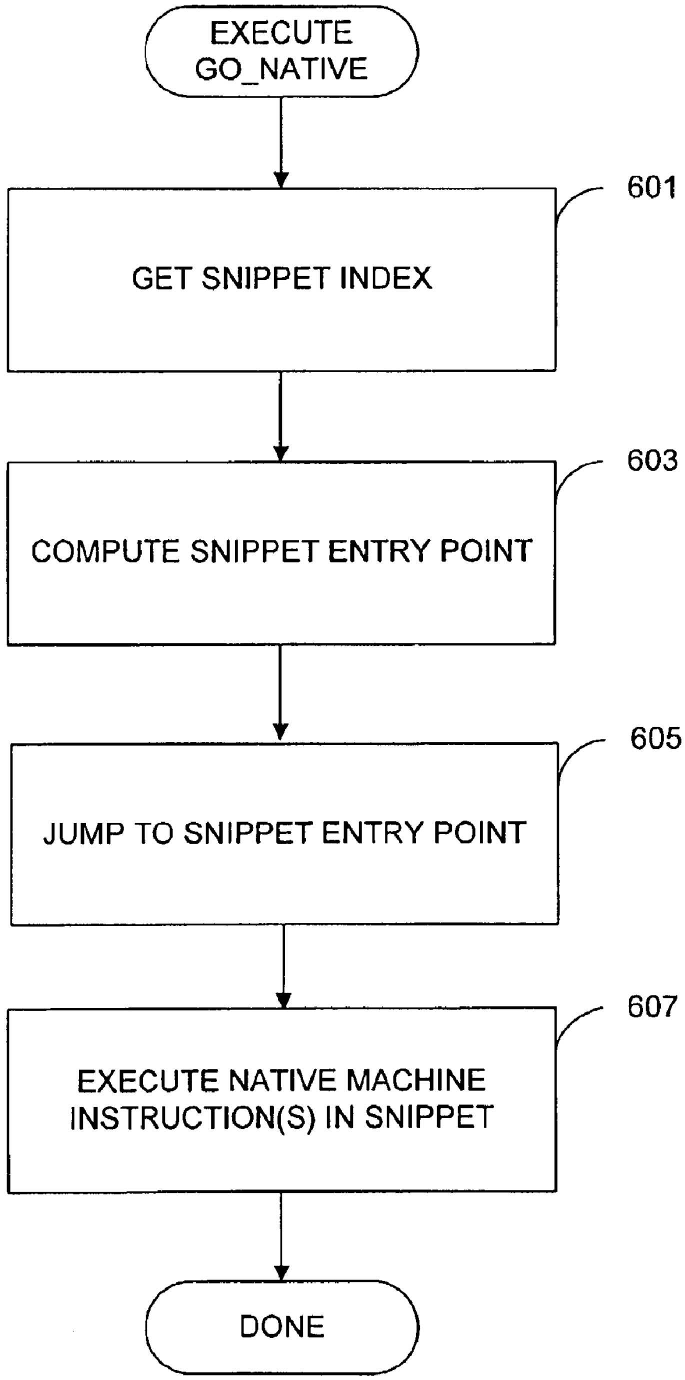

EXECUTE GO_NATIVE
601
GET SNIPPET INDEX
603
COMPUTE SNIPPET ENTRY POINT
605
JUMP TO SNIPPET ENTRY POINT
607
EXECUTE NATIVE MACHINE INSTRUCTION(S) IN SNIPPET
DONE

FIG. 8

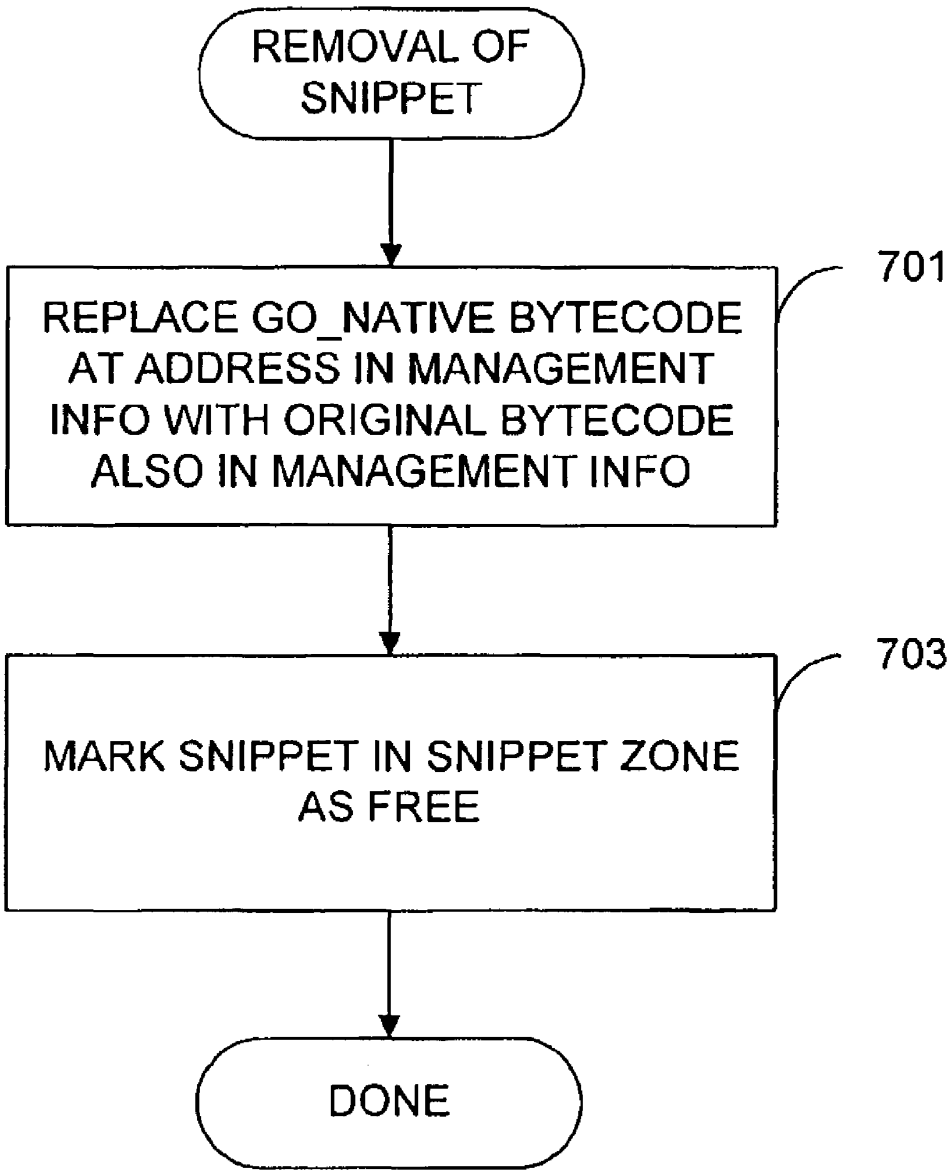

REMOVAL OF SNIPPET
REPLACE GO_NATIVE BYTECODE AT ADDRESS IN MANAGEMENT INFO WITH ORIGINAL BYTECODE ALSO IN MANAGEMENT INFO
701
MARK SNIPPET IN SNIPPET ZONE AS FREE
703
DONE

FIG. 9

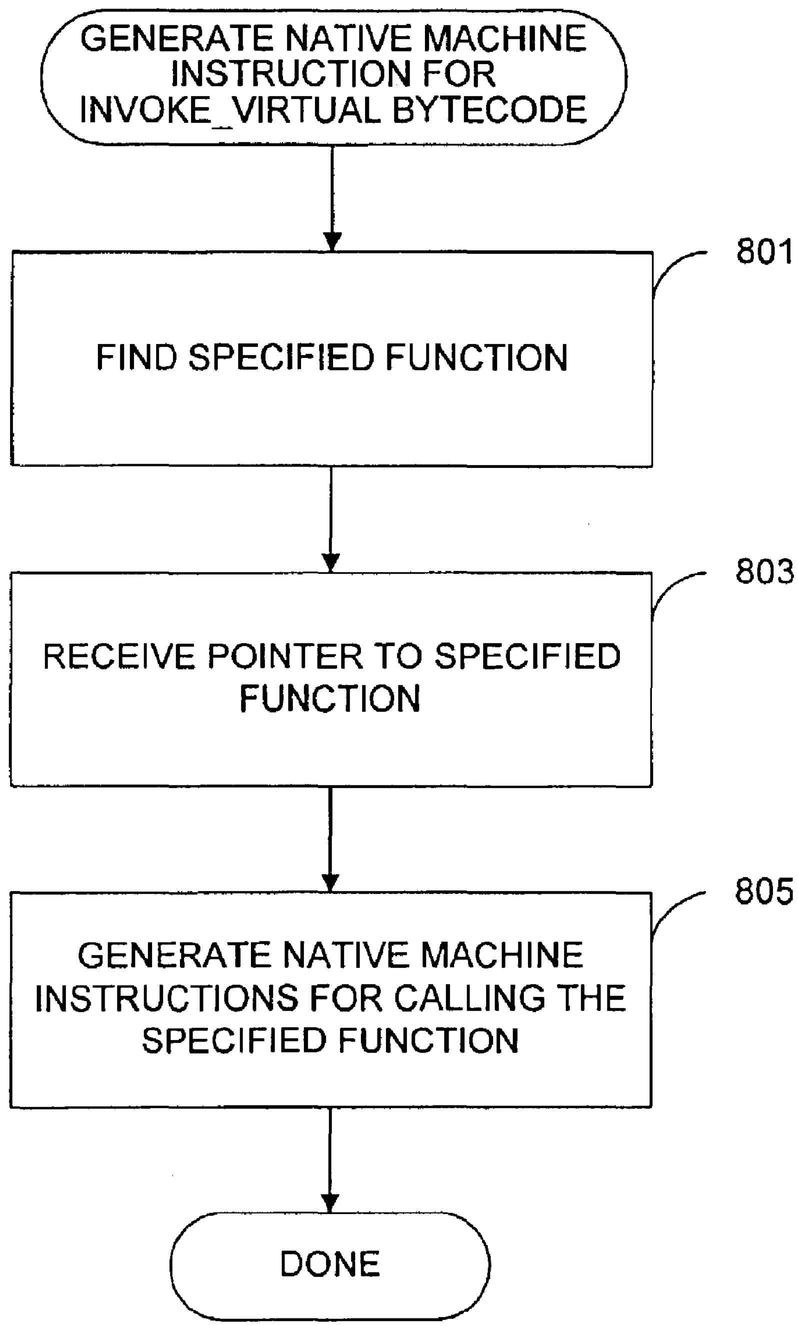

GENERATE NATIVE MACHINE INSTRUCTION FOR INVOKE_VIRTUAL BYTECODE
801
FIND SPECIFIED FUNCTION
803
RECEIVE POINTER TO SPECIFIED FUNCTION
805
GENERATE NATIVE MACHINE INSTRUCTIONS FOR CALLING THE SPECIFIED FUNCTION
DONE

FIG. 10

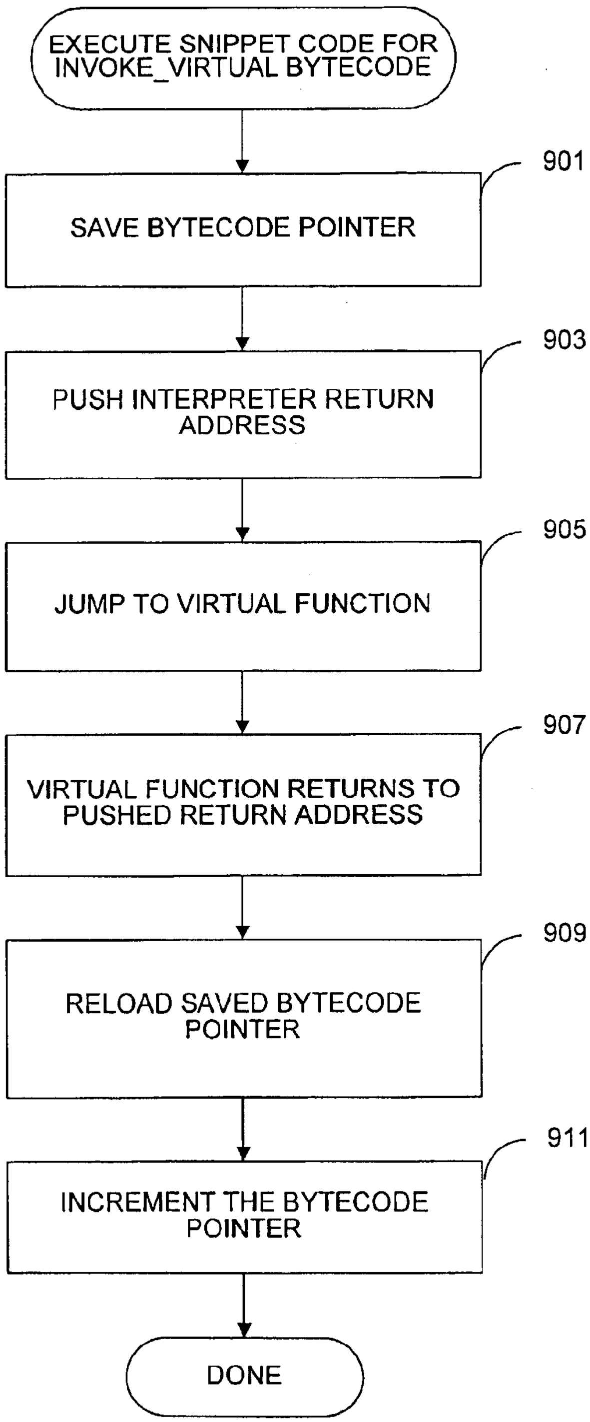

EXECUTE SNIPPET CODE FOR INVOKE_VIRTUAL BYTECODE
SAVE BYTECODE POINTER
901
PUSH INTERPRETER RETURN ADDRESS
903
JUMP TO VIRTUAL FUNCTION
905
VIRTUAL FUNCTION RETURNS TO PUSHED RETURN ADDRESS
907
RELOAD SAVED BYTECODE POINTER
909
INCREMENT THE BYTECODE POINTER
911
DONE

FIG. 11

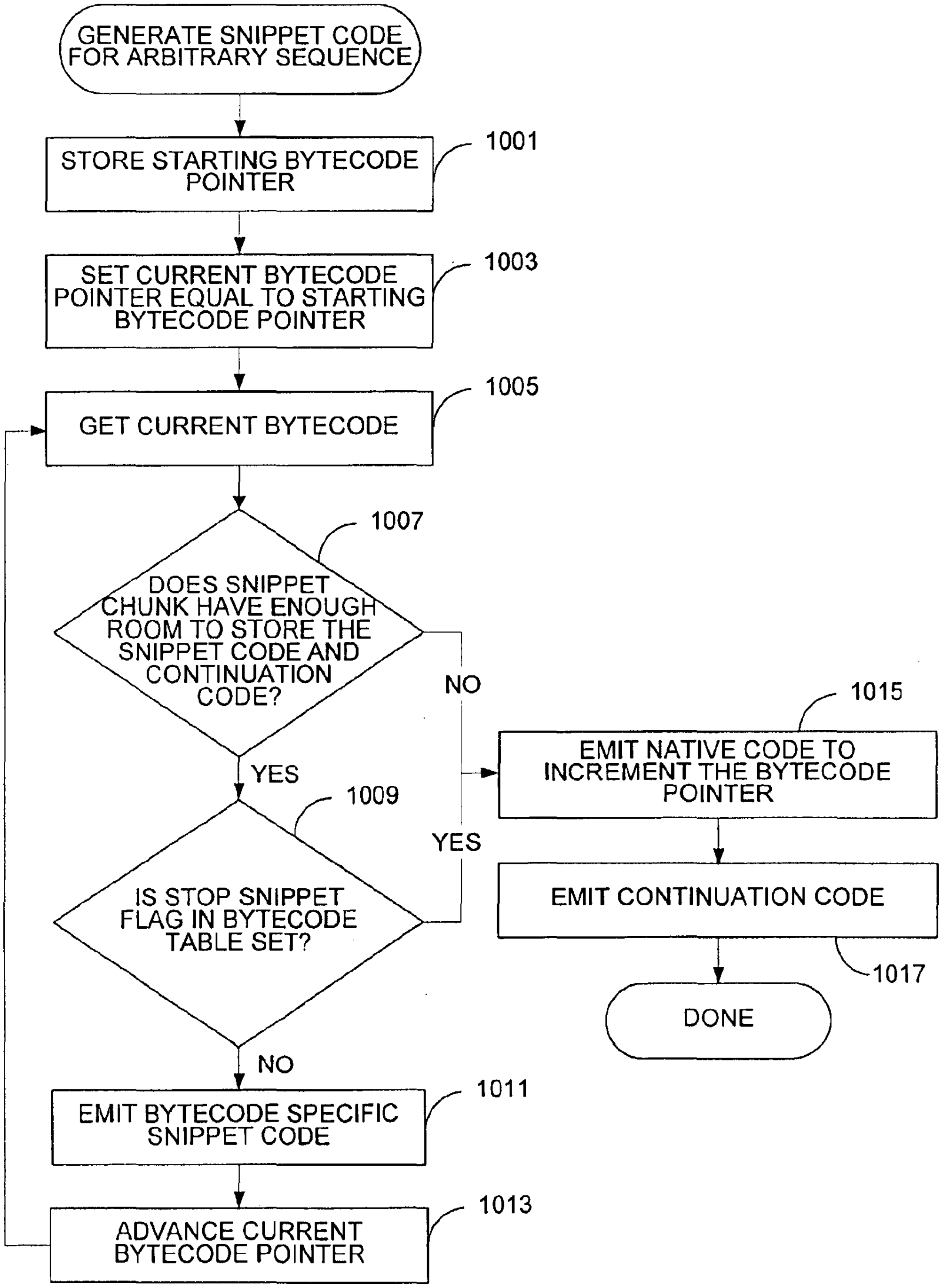

GENERATE SNIPPET CODE FOR ARBITRARY SEQUENCE
STORE STARTING BYTECODE POINTER
1001
SET CURRENT BYTECODE POINTER EQUAL TO STARTING BYTECODE POINTER
1003
GET CURRENT BYTECODE
1005
DOES SNIPPET CHUNK HAVE ENOUGH ROOM TO STORE THE SNIPPET CODE AND CONTINUATION CODE?
1007
NO
YES
IS STOP SNIPPET FLAG IN BYTECODE TABLE SET?
1009
YES
NO
EMIT BYTECODE SPECIFIC SNIPPET CODE
1011
ADVANCE CURRENT BYTECODE POINTER
1013
EMIT NATIVE CODE TO INCREMENT THE BYTECODE POINTER
1015
EMIT CONTINUATION CODE
1017
DONE

FIG. 12

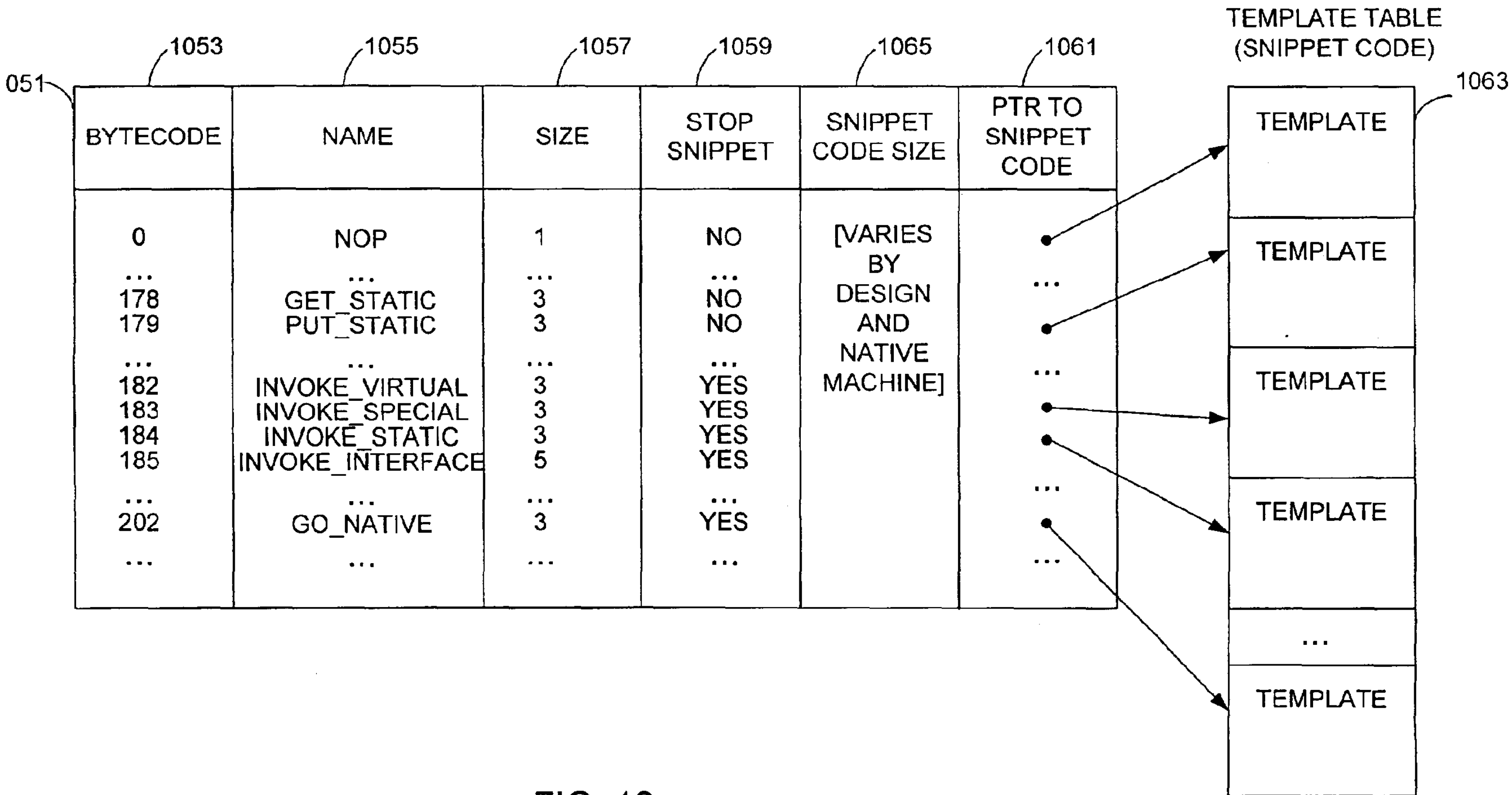

TEMPLATE TABLE (SNIPPET CODE)
1053
1055
1057
1059
1065
1061
1063
051
BYTECODE
NAME
SIZE
STOP SNIPPET
SNIPPET CODE SIZE
PTR TO SNIPPET CODE
0
NOP
1
NO
[VARIES BY DESIGN AND NATIVE MACHINE]
178
GET_STATIC
3
NO
179
PUT_STATIC
3
NO
182
INVOKE_VIRTUAL
3
YES
183
INVOKE_SPECIAL
3
YES
184
INVOKE_STATIC
3
YES
185
INVOKE_INTERFACE
5
YES
202
GO_NATIVE
3
YES
TEMPLATE
TEMPLATE
TEMPLATE
TEMPLATE
TEMPLATE

FIG. 13

INTERPRETING FUNCTIONS UTILIZING A HYBRID OF VIRTUAL AND NATIVE MACHINE INSTRUCTIONS

This is a Continuation application of prior application Ser. No. 08/884,856 filed on Jun. 30, 1997, now U.S. Pat. No. 6,513,156 the disclosure of which is incorporated herein by reference.

BACKGROUND OF THE INVENTION

The present invention relates to increasing the execution speed of interpreters and, more specifically, increasing the speed of an interpreter executing a Java™ function utilizing a hybrid of virtual and native machine instructions.

The computer era began back in the early 1950s with the development of the UNIVAC. Today, there are countless numbers of computers and computer platforms. Although the variety of computers is a blessing for users, it is a curse for programmers and application developers who have the unfortunate task of modifying or porting an existing computer program to run on a different computer platform.

One of the goals of high level languages is to provide a portable programming environment such that the computer programs may be easily ported to another computer platform. High level languages such as "C" provide a level of abstraction from the underlying computer architecture and their success is well evidenced from the fact that most computer applications are now written in a high level language.

Portability has been taken to new heights with the advent of World Wide Web ("the Web") which is an interface protocol for the Internet which allows communication of diverse computer platforms through a graphical interface. Computers communicating over the Web are able to download and execute small applications called applets. Given that applets may be executed on a diverse assortment of computer platforms, the applets are typically executed by a Java™ virtual machine.

The Java™ programming language is an object-oriented high level programming language developed by Sun Microsystems and designed to be portable enough to be executed on a wide range of computers ranging from small personal computers up to supercomputers. Java programs are compiled into class files which include virtual machine instructions (e.g., bytecodes) for the Java virtual machine. The Java virtual machine is a software emulator of a "generic" computer. An advantage of utilizing virtual machine instructions is the flexibility that is achieved since the virtual machine instructions may be run, unmodified, on any computer system that has a virtual machine implementation, making for a truly portable language. Additionally, other programming languages may be compiled into Java virtual machine instructions and executed by a Java virtual machine.

The Java virtual machine is an interpreter executed as an interpreter program (i.e., software). Conventional interpreters decode and execute the virtual machine instructions of an interpreted program one instruction at a time during execution. Compilers, on the other hand, decode source code into native machine instructions prior to execution so that decoding is not performed during execution. Because conventional interpreters decode each instruction before it is executed repeatedly each time the instruction is encountered, execution of interpreted programs is typically quite slower than compiled programs because the native machine instructions of compiled programs can be executed on the native machine or computer system without necessitating decoding.

A known method for increasing the execution speed of Java interpreted programs of virtual machine instructions involves utilizing a just-in-tine (JIT) compiler. The JIT compiler compiles an entire Java function just before it is called. However, native code generated by a JIT compiler does not always run faster than code executed by an interpreter. For example, if the interpreter is not spending the majority of its time decoding the Java virtual machine instructions, then compiling the instructions with a JIT compiler may not increase the execution speed. In fact, execution may even be slower utilizing the JIT compiler if the overhead of compiling the instructions is more than the overhead of simply interpreting the instructions.

Another known method for increasing the execution speed of Java interpreted programs of virtual machine instructions utilizes "quick" instructions or bytecodes. The "quick" instructions take advantage of the unassigned bytecodes for the Java virtual machine. A "quick" instruction utilizes an unassigned bytecode to shadow another bytecode. The first time that the shadowed bytecode is encountered, the bytecode is replaced by the "quick" bytecode which is a more efficient implementation of the same operation. Although "quick" instructions have been implemented with good results, their flexibility is limited since the number of unassigned bytecodes is limited (and may decrease as new bytecodes are assigned).

Accordingly, there is a need for new techniques for increasing the execution speed of computer programs that are being interpreted. Additionally, there is a need for new techniques that provide flexibility in the way in which interpreted computer programs are executed.

SUMMARY OF THE INVENTION

In general, embodiments of the present invention provide innovative systems and methods for increasing the execution speed of computer programs executed by an interpreter. A portion of a function is compiled into at least one native machine instruction so that the function includes both virtual and native machine instructions during execution. With the invention, the mechanism for increasing the execution speed of the virtual machine instructions is transparent to the user, the hybrid virtual and native machine instructions may be easily transformed back to the original virtual machine instructions, and the flexibility of compiling only certain portions of a function into native machine instructions allows for better optimization of the execution of the function. Several embodiments of the invention are described below.

In one embodiment, a computer implemented method for increasing the execution speed of virtual machine instructions is provided. Virtual machine instructions for a function are input into a computer system. A portion of the function is compiled into native machine instruction(s) so that the function includes both virtual and native machine instructions. A virtual machine instruction of the function may be overwritten with a new virtual machine instruction that specifies the execution of native machine instructions that were compiled from a sequence of virtual machine instructions beginning with the overwritten virtual machine instruction of the function. In preferred embodiments, the virtual machine instructions are Java virtual machine instructions.

In another embodiment, a computer implemented method for increasing the execution speed of virtual machine instructions is provided. Java virtual machine instructions for a function are input into a computer system. A portion of the function is compiled into native machine instruction(s).

A copy of a selected virtual machine instruction at a beginning of the portion of the function is stored and a back pointer to a location of the selected virtual machine instruction is also stored. The selected virtual machine instruction is overwritten with a new virtual machine instruction that specifies execution of the native machine instructions so that the function includes both virtual and native machine instructions. The new virtual machine instruction may include a pointer to a data block in which is stored the native machine instructions, the copy of the selected virtual machine instruction, and the back pointer. Additionally, the original virtual machine instructions that were input may be generated by storing the copy of the selected virtual machine instruction stored in the data block at the location specified by the back pointer.

In another embodiment, a computer implemented method of generating hybrid virtual and native machine instructions is provided. A sequence of virtual machine instructions for a function is input into a computer system. A virtual machine instruction of the sequence of virtual machine instructions is selected and the selected virtual machine instruction is overwritten with a new virtual machine instruction that specifies one or more native machine instructions. The new virtual machine instruction may include a pointer to the one or more native machine instructions which may be stored in a data block. The one or more native machine instructions may be generated from a compilation of a portion of the sequence of virtual machine instructions beginning with the selected virtual machine instruction.

Other features and advantages of the invention will become readily apparent upon review of the following detailed description in association with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates an example of a computer system that may be utilized to execute the software of an embodiment of the invention.

FIG. **2** shows a system block diagram of the computer system of FIG. **1**.

FIG. **3** shows how a Java source code program is executed.

FIG. **4** shows a high level flowchart illustrating a process of transforming a function into a hybrid of virtual and native machine instructions in accordance with one embodiment of the present invention.

FIG. **5** illustrates a transformation of Java virtual machine instructions of a function to hybrid virtual and native machine instructions.

FIG. **6** shows a process of introducing snippets which are native machine instructions compiled from a sequence of virtual machine instructions of a function.

FIG. **7** shows a process of allocating a snippet in the snippet zone which stores all existing snippets.

FIG. **8** shows a process of executing a go_native virtual machine instruction that specifies the execution of native machine instructions in a snippet.

FIG. **9** shows a process of removing a snippet from the hybrid, virtual, and native machine instructions of a function.

FIG. **10** shows a process of generating native machine instructions for the invoke_virtual bytecode.

FIG. **11** shows a process of executing snippet code for the invoke_virtual bytecode.

FIG. **12** shows a process of generating snippet code for an arbitrary sequence of virtual machine instructions in a function.

FIG. **13** illustrates a bytecode table which may be utilized to store information regarding different Java bytecodes.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Definitions

Machine instruction—An instruction that directs a computer to perform an operation specified by an operation code (OP code) and optionally one or more operand.

Virtual machine instruction—A machine instruction for a software emulated microprocessor or computer architecture (also called virtual code).

Native machine instruction—A machine instruction that is designed for a specific microprocessor or computer architecture (also called native code).

Class—An object-oriented data type that defines the data and methods that each object of a class will include.

Function—A software routine (also called a subroutine, procedure, member and method).

Snippet—A relatively small piece of compiled native machine instructions and associated information.

Bytecode pointer (BCP)—A pointer that points to the current Java virtual machine instruction (e.g., bytecode) that is being executed.

Program counter (PC)—A pointer that points to the machine instruction of the interpreter that is being executed.

Overview

In the description that follows, the present invention will be described in reference to a preferred embodiment that increases the execution speed of Java virtual machine instructions. However, the invention is not limited to any particular language, computer architecture, or specific implementation. As an example, the invention may be advantageously applied to languages other than Java (e.g., Smalltalk). Therefore, the description of the embodiments that follow is for purposes of illustration and not limitation.

FIG. **1** illustrates an example of a computer system that may be used to execute the software of an embodiment of the invention. FIG. **1** shows a computer system **1** which includes a display **3**, screen **5**, cabinet **7**, keyboard **9**, and mouse **11**. Mouse **11** may have one or more buttons for interacting with a graphical user interface. Cabinet **7** houses a CD-ROM drive **13**, system memory and a hard drive (see FIG. **2**) which may be utilized to store and retrieve software programs incorporating computer code that implements the invention, data for use with the invention, and the like. Although the CD-ROM **15** is shown as an exemplary computer readable storage medium, other computer readable storage media including floppy disk, tape, flash memory, system memory, and hard drive may be utilized. Additionally, a data signal embodied in a carrier wave (e.g., in a network including the Internet) may be the computer readable storage medium.

FIG. **2** shows a system block diagram of computer system **1** used to execute the software of an embodiment of the invention. As in FIG. **1**, computer system **1** includes monitor **3** and keyboard **9**, and mouse **11**. Computer system **1** further includes subsystems such as a central processor **51**, system memory **53**, fixed storage **55** (e.g., hard drive), removable storage **57** (e.g., CD-ROM drive), display adapter **59**, sound card **61**, speakers **63**, and network interface **65**. Other computer systems suitable for use with the invention may include additional or fewer subsystems. For example, another computer system could include more than one processor **51** (i.e., a multi-processor system), or a cache memory.

The system bus architecture of computer system **1** is represented by arrows **67**. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system **1** shown in FIG. **2** is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

Typically, computer programs written in the Java programming language are compiled into bytecodes or Java virtual machine instructions which are then executed by a Java virtual machine. The bytecodes are stored in class files which are input into the Java virtual machine for interpretation. FIG. **3** shows a progression of a simple piece of Java source code through execution by an interpreter, the Java virtual machine.

Java source code **101** includes the classic Hello World program written in Java. The source code is then input into a bytecode compiler **103** which compiles the source code into bytecodes. The bytecodes are virtual machine instructions as they will be executed by a software emulated computer. Typically, virtual machine instructions are generic (i.e., not designed for any specific microprocessor or computer architecture) but this is not required. The bytecode compiler outputs a Java class file **105** which includes the bytecodes for the Java program.

The Java class file is input into a Java virtual machine **107**. The Java virtual machine is an interpreter that decodes and executes the bytecodes in the Java class file. The Java virtual machine is an interpreter, but is commonly referred to as a virtual machine as it emulates a microprocessor or computer architecture in software (e.g., the microprocessor or computer architecture that may not exist).

An interpreter may execute a bytecode program by repeatedly executing the following steps:

Execute—execute operation of the current bytecode

Advance—advance bytecode pointer to next bytecode

Dispatch—fetch the bytecode at the bytecode pointer and jump to the implementation (i.e., execute step) of that bytecode.

The execute step implements the operation of a particular bytecode. The advance step increments the bytecode pointer so that it points to the next bytecode. Lastly, the dispatch step fetches the bytecode at the current bytecode pointer and jumps to the piece of native machine code that implements that bytecode. The execution of the execute-advance-dispatch sequence for a bytecode is commonly called an "interpretation cycle."

Although in a preferred embodiment, the interpreter utilizes the interpretation cycle described above. Many other interpretation cycles may be utilized in conjunction with the present invention. For example, an interpreter may perform dispatch-execute-advance interpretation cycles or there may be more or fewer steps in each cycle. Accordingly, the invention is not limited to the embodiments described herein.

Hybrid Virtual and Native Machine Instructions

In general, the speed in which a program is interpreted can be increased by reducing the average time needed for an interpretation cycle. The invention recognizes the fact that on many modern computers the dispatch step is often the most time consuming step. Accordingly, the advance and dispatch steps of several bytecodes may be combined into a single advance and dispatch step which significantly decreases the execution time needed for such a bytecode sequence.

As an example, assume that the Java source code statement X:=A+B was compiled into bytecodes represented by the following instructions:

Load A

2. Load B

3. Add

4. Store X

The Java virtual machine is a stack based machine. Therefore, after the values of A and B are loaded onto the stack, these values are added and removed from the stack with the result being placed on the stack. The result on the stack is then stored in X and the result is removed from the stack.

When the above virtual machine instructions are interpreted, each of the execute, advance and dispatch steps will take a certain amount of time which may vary from instruction to instruction. Accordingly, the time it takes the interpreter to execute an instruction will be the sum of the time it takes to execute each of the execute, advance and dispatch steps for that instruction. The time it takes to execute a step will be represented by E_n for execute, A_n for advance and D_n for dispatch, where the subscript indicates the number of the instruction to which the time is associated.

The time it takes the interpreter to execute the virtual machine instructions shown above will be the sum of the following times which may occur in this order: E_1, A_1, D_1, E_2, A_2, D_2, E_3, A_3, D_3, E_4, A_4, D_4. With an embodiment of the present invention, a sequence of virtual machine instructions as shown above may be compiled into native machine instructions in the form of a "snippet" so that all but the last advance and dispatch steps may be removed.

As a snippet includes a sequence of native machine instructions, the advance and dispatch steps between the instructions may be eliminated. Therefore, the execution time of the snippet will be approximately the sum of the following times in this order: E_1, E_2, E_3, E_4, A_S, D_S, where the subscript "S" indicates these times represent the snippet advance and dispatch steps which may be different than the traditional advance and dispatch steps. Since the initial advance and dispatch steps are no longer needed to advance the bytecode pointer and fetch the next bytecode, the snippet includes an optimized interpretation cycle for a sequence of bytecodes while preserving interpreter semantics. Conceptually, therefore, a snippet may be considered as an implementation of a higher level bytecode that implements the operations of a sequence of lower level bytecodes.

FIG. **4** shows a high level flowchart of a process of generating a hybrid of virtual and native machine instructions for a function in accordance with one embodiment of the present invention. At step **201**, virtual machine instructions for a function are input into a computer system, such as the ones shown in FIGS. **1** and **2**. In preferred embodiments, the virtual machine instructions for a function are stored as a class file of bytecodes. However, the invention may be readily extended into other interpreted languages by an extension of the principles described herein.

A portion of the virtual machine instructions of the function is selected to be compiled at step **203**. Typically, the system recognizes individual bytecodes or sequences of bytecodes that may be advantageously compiled. For example, the system may generate a snippet for each Java invoke_virtual bytecode that is encountered. Since the invoke_virtual op code may be optimized when it is compiled into native machine instructions within a snippet (see also the section entitled "In-line Caching"). Additionally, statistics may be collected during the interpretation of a program in order to identify portions of the program that would benefit from having a snippet generated.

At step **205**, the selected portion of the function is compiled into one or more native machine instructions. Although snippets usually include more than one native machine instruction, the number of machine instructions is dependent on the virtual machine instructions for which the snippet is replacing.

The virtual machine instruction at the beginning of the selected portion of the function is saved at step **207**. It is not required in all instances that the entire virtual machine instruction be saved. For example, in some embodiments only an initial portion (e.g., first one or more bytes) of a virtual machine instruction are saved. Therefore, when it is stated that a virtual machine instruction is saved, it should be understood that it is meant that at least an initial portion of the virtual machine instruction is saved. Furthermore, in some embodiments more than one virtual machine instruction at the beginning of the selected portion of the function may be saved. It will be readily understood by those of skill in the art that the number of bytes or virtual machine instructions that are saved (or overwritten) may be varied in different embodiments and may depend on the virtual machine instructions themselves.

In order for the snippet to be executed, a new virtual machine instruction (called "go_native" in a preferred embodiment) is executed which specifies the subsequent execution of the snippet. This new virtual machine instruction replaces or overwrites the initial virtual machine instruction of the selected portion of the function. So that the original function or computer program may be restored, the original virtual machine instruction at the beginning of the selected portion is saved, at step **207**, prior to being overwritten. This process will be described in more detail upon reference to FIG. **5** which illustrates how Java virtual machine instructions of a function may be transformed into hybrid virtual and native machine instructions.

At step **209**, the virtual machine instruction at the beginning of the selected portion of the function is overwritten with a new virtual machine instruction that specifies the execution of the one or more native machine instructions of the snippet. In the Java virtual machine, the virtual machine instructions are bytecodes meaning that each virtual machine instruction is composed of one or more bytes. The examples described herein refer to preferred embodiments which increase the execution speed of programs for the Java virtual machine. However, the invention may be advantageously applied to other interpreted languages where the virtual machine instructions may not necessarily be bytecodes.

During execution of an interpreted program, the interpreter decides when to substitute a sequence of bytecodes with a snippet. In a preferred embodiment, if a sequence of bytecodes which may be replaced by a snippet has been found, the interpreter generates a snippet for the sequence and then overwrites the first three bytes of that sequence with a go_native bytecode and a two byte number specifying the snippet. The go_native bytecode is an unused bytecode which is selected for use of the invention.

The snippet will not only hold the native machine instructions, but also the three bytes of the original bytecode that was overwritten as well as a pointer back to their original location so that the snippet may be removed and the original bytecodes restored.

FIG. **5** shows a generation of hybrid virtual and native machine instructions. Java virtual machine instructions **301** are bytecodes where each bytecode may include one or more bytes. The Java virtual machine instructions typically reside in a Java class file as is shown in FIG. **3**. In the example shown, the interpreter decides to introduce a snippet for bytecodes **2**–**5** of virtual machine instructions **301**. The interpreter generates modified Java virtual machine instructions **303** by overwriting bytecode **2** with a go_native virtual machine instruction.

A snippet zone **305** stores snippets which include native machine instructions. As shown, the go_native bytecode includes a pointer or index to a snippet **307**. Each snippet is a data block that includes two sections of which the first is management information and the second is a sequence of one or more native machine instructions. The management information includes storage for the original bytecode **2** which was overwritten by the go_native bytecode and also the original address of bytecode **2** so that the original bytecode sequence may be restored when the snippet is removed. Typically, the management information section of the snippet is of a fixed length so that the native machine instructions may be easily accessed by a fixed offset. Although snippet **307** is shown as occupying a single "chunk" in snippet zone **305**, snippets may also be allocated that occupy more than one chunk of the snippet zone.

The native machine instruction section of snippet **307** includes native machine instructions for bytecodes **2**–**5** of virtual machine instructions **301**. Hybrid virtual and native machine instructions **309** include the modified virtual machine instructions and the snippets in the snippet zone. When the interpreter executes the go_native bytecode, the interpreter will look up the snippet in the snippet zone specified by the go_native bytecode and then activate the native machine instructions in the snippet.

The native machine instructions in the snippet perform the same operations as if the bytecodes **2**–**5** would have been interpreted. Afterwards, the interpreter continues with the execution of bytecode **6** as if no snippet existed. The return of execution in virtual machine instructions **301** is indicated by the dashed arrow in hybrid virtual and native machine instructions **309** shown in FIG. **5**.

The go_native bytecode references (e.g., has a pointer to) the snippet and the snippet includes a reference to the location of the go_native bytecode. The go_native bytecode in a preferred embodiment is 3 bytes long: one for the go_native op code and two bytes for an index into the snippet zone. The two-byte index allows for over 65,000 snippets in the snippet zone. Within the snippet management information section is stored the address of the original bytecode which is currently occupied by the go_native bytecode. This address is utilized to write the original bytecode also stored in the management information section back to its original location. Although a preferred embodiment utilizes a three byte go_native bytecode, there is no requirement that this size be utilized. For example, any number of bytes may be utilized or the size does not have to be limited to byte boundaries.

Snippets should be introduced selectively because it takes time to generate the snippets and because the snippets consume memory space. In a preferred embodiment, snippets are introduced for the following: all or portions of loop bodies, special Java bytecodes (e.g., get_static and put_static), and Java message sends (all the invokexxx bytecodes). In the bytecodes for Java virtual machine, loops are implemented using backward branches. Thus, whenever the interpreter encounters a backward branch, it may introduce a snippet. The snippet generator generates as much native code that will fit into the snippet, starting with the backward branch bytecode. Additionally, some special Java bytecodes and Java message sends may be sped up by using snippets.

FIG. **6** shows a process of introducing a snippet. At step **401**, the system allocates a free snippet in the snippet zone. The free snippet is storage space within the snippet zone which has not been utilized or has been marked as available. One process of allocating a free snippet will be described in more detail in reference to FIG. **7**.

Once a free snippet has been obtained, the one or more virtual machine instructions are compiled into one or more native machine instructions at step **403**. Although the flowcharts show an order to the steps, no specific ordering of the steps should be implied from the figures. For example, it is not necessary that a free snippet be allocated before the virtual machine instructions are compiled into native machine instructions. In fact, in some embodiments it may be beneficial to compile the virtual machine instructions first and then allocate a free snippet to store the native machine instructions, especially if a snippet may span more than one chunk in the snippet zone.

At step **405**, a copy of a selected virtual machine instruction is saved in the management information section of the allocated snippet. The selected virtual machine instruction is the virtual machine instruction that was originally at the beginning of the compiled virtual machine instructions of a function. However, in some embodiments only an initial portion (one or more bytes) of the original virtual machine instruction is saved in the snippet. The address of the original virtual machine instruction in the function is saved in the management information section of the allocated snippet at step **407**.

At step **409**, the original virtual machine instruction is overwritten with a new virtual machine instruction (e.g., go_native) that points to the allocated snippet. As snippets are generated during program execution, the new virtual machine instruction is executed at step **411**.

A snippet may be introduced at arbitrary locations in a bytecode program. However, if the go_native bytecode spans across more than one of the original bytecodes it should be verified that the second and subsequent original bytecodes overwritten by the go_native bytecode are not jump targets or subroutine entry points. More generally, a go_native bytecode should not be used across a basic block entry point. Nevertheless, backward branches as well as many other Java bytecodes are at least three bytes long, thereby providing plenty of storage space for a go_native bytecode. It should be mentioned that a jump to a bytecode after the go_native bytecode which has been compiled into a snippet will not present any problems since the bytecode remains untouched at that location.

Snippets are held and managed in a separate memory space called the snippet zone. The snippet zone may be thought of as a circular list of snippets where a snippet is allocated by either finding an unused snippet in the snippet zone or by recycling a used snippet. Preferably all snippets have the same size to simplify management of the snippet zone. In general, the more snippets that are present in the snippet zone, the longer it will take before a snippet has to be recycled and therefore the faster the program will run.

Now that it has been shown how a snippet may be introduced, FIG. **7** shows a process of allocating a snippet in the snippet zone which was shown as step **401** in FIG. **6**. The process shown in FIG. **7** utilizes a round robin fashion to allocate snippets (i.e., as soon as a new snippet is needed and there are no unused snippets left, the next snippet in the circular list of snippets of the snippet zone is recycled).

At step **501**, the system gets the current snippet. The current snippet in the snippet zone is indicated by a snippet pointer. The system determines if the current snippet is free to be used at step **503**. A flag may be present in the management information section of the snippet to indicate whether the snippet is available. In some embodiments, the field in the management information section of the snippet which stores the address of the original bytecode is set to null if the snippet is available.

If the current snippet is not free, the current snippet is removed at step **505**. Removing the current snippet includes writing the original bytecode stored in the management information section of the snippet to the address of the original bytecode also stored in the management section of the snippet. A process of removing a snippet will be described in more detail in reference to FIG. **9**.

After a snippet has been allocated in the snippet zone, the allocated snippet is set equal to the current snippet, at step **507**, since now the current snippet is free. At step **509**, the system increments the snippet pointer. Although the snippet zone may be thought of as a circular list, the snippet zone may be implemented as an array of chunks. Therefore, if the snippet zone is a linear array, incrementing the snippet pointer may also involve resetting the snippet pointer to the beginning of the snippet zone if the snippet pointer has passed the end of the snippet zone.

Another approach to managing snippets in the snippet zone is to use a time stamp that is stored in the management information section of the snippet indicating the time when the snippet was created or last used. Since it may take substantial resources to find the snippet with the oldest time stamp to be recycled, a combination of time stamps and the round robin fashion may be utilized as follows.

When a free snippet is required, the system may search a predetermined number of snippets after the snippet pointer (e.g., 5 or 10 snippets) in order to locate a snippet with an old time stamp. The snippet with the oldest time stamp near the snippet pointer may then be recycled. Additionally, the time stamp field in the management information section of the snippet may be set to zero or an old time stamp in order to mark the snippet as free.

Now that it has been shown how a snippet may be set up, FIG. **8** shows a process of executing a go_native bytecode. At step **601**, the system gets the snippet index or pointer from the go_native bytecode. The snippet index may be a 2 byte offset into the snippet zone. The system computes the snippet entry point of the native machine instructions within the snippet at step **603**. The snippet entry point is the location of the native machine instructions after the management information section of the snippet. Since the management information section is typically a fixed size, calculating the snippet entry point typically includes adding an offset to the address of the snippet.

The system then jumps to the snippet entry point at step **605** in order to begin execution of the native machine instructions of the snippet. The native machine instructions in the snippet are executed in a step **607**.

Although the implementation of snippets increases the speed of execution of the interpreted code, it is also desirable to provide the capability to reverse the introduction of snippets in order to generate the original bytecodes. For example, after a program in memory has executed, it may be desirable to generate a listing of the original bytecodes without requiring that the original class files be available for access.

FIG. **9** shows a process of removing a snippet in order to produce the original bytecodes. At step **701**, the system replaces the go_native bytecode at the address stored in the management information section of the snippet with the original bytecode (or its initial bytes) also stored in the

management information section. The address stored in the management information section acts as a back pointer to the original bytecode.

Once the original bytecodes are restored, the snippet may be marked as free in the snippet zone at step **703**. The snippet may be marked free in any number of ways depending upon the implementation of the snippet zone. For example, a null pointer may be stored in the address of the original bytecode within the management information section of the snippet. Additionally, if time stamps are being utilized, the time stamp may be set to zero or an old value in order to mark the snippet as free in the snippet zone.

The preceding has described how the invention utilizes dynamically generated native machine instructions for sequences of interpreted code so that a function may be more efficiently executed utilizing a hybrid of virtual and native machine instructions. The execution of an interpreted program can be significantly sped up because frequently used code sequences may be executed in native code rather than an interpreted fashion. The snippets generated are transparent to the interpreter and impose no additional states or complexity. The following will describe implementations of specific virtual machine instruction situations.

In-Line Caching

In the Java virtual machine, the invoke_virtual bytecode is utilized to invoke "normal" functions. The invoke_virtual bytecode includes two bytes which, among other things, specify a function to be invoked. During interpretation of the invoke_virtual bytecode, the interpreter first decodes and executes the invoke_virtual bytecode. The execution of the invoke_virtual bytecode involves fetching the two bytes and determining the starting address of the specified function. However, the determination of the starting address of the specified function may include following multiple levels of pointers to find the class that includes the function. Consequently, the interpretation of an invoke_virtual bytecode maybe be very time consuming.

Snippets may be utilized to expedite the execution of the invoke_virtual bytecode by compiling the invoke_virtual bytecode into the native machine instruction equivalent of "call <function>" (i.e., the starting address of the function is specified without requiring a time consuming search for the starting address of the function). FIG. **10** shows a process of generating a native machine instruction for the invoke_virtual bytecode.

At step **801**, the system finds the function specified in the invoke_virtual bytecode. The process for finding the specified may be the same as is executed by an interpreter (e.g., pointers from class definitions will be followed to find the specified function). Once the specified function is found, the system receives a pointer or address to the specified virtual function at step **803**.

The system then generates native machine instructions for calling the specified virtual function at step **805**. The native machine instructions include the address of the specified function so that execution of the invoke_virtual bytecode will no longer necessitate the time consuming process of finding the starting address of the specified function. By "hard coding" the address of the desired function in native machine instruction, there is a substantial increase in the speed of execution of the invoke_virtual bytecode.

Now that it has been described how the go_native bytecode for implementing the invoke_virtual bytecode has been set up, FIG. **11** shows a process of executing snippet code for the invoke_virtual bytecode. At step **901**, the system saves the current bytecode pointer so that the interpreter can continue at the right location after returning from the function invoked by the invoke_virtual bytecode.

The system pushes the interpreter return address on the stack at step **903**. The interpreter return address is a predefined location where the execution of the interpreter from invoke_virtual bytecodes should resume. The native machine instructions in the snippet for the invoke_virtual function then instruct the system to jump to the function specified in the invoke_virtual bytecodes at step **905**.

Once the virtual function finishes execution, the system returns to the return address that was pushed on the stack at step **907**. At the return address, there are native machine instructions for the interpreter to reload the saved bytecode pointer. At step **909**, recalling that the bytecode pointer was saved at step **901**, the system reloads the saved bytecode pointer so the interpreter may continue where it left off. The interpreter increments the bytecode pointer, at step **909**, in order to indicate the bytecode that should be interpreted next.

As shown above, snippets may be utilized to increase the execution performance of the invoke_virtual bytecode. Other Java bytecodes may be similarly optimized including the invoke_static, invoke_interface, and invoke_special.

Arbitrary Sequences

As described earlier, snippets may be generated for arbitrary sequences of virtual machine instructions. The arbitrary sequences of virtual machine instructions may be selected any number of ways including a statistical analysis that determines execution speed will be increased upon snippetization of the identified sequence of virtual machine instructions.

FIG. **12** shows a process for generating snippet code for an arbitrary sequence of virtual machine instructions. At step **1001**, the system stores the starting bytecode pointer. The starting bytecode pointer indicates the first bytecode that is represented by the snippet that will be generated. At step **1003**, the system sets the current bytecode pointer equal to the starting bytecode pointer. The current bytecode pointer will be utilized to "walk through" the bytecodes as they are compiled and placed in the snippet. The system gets the current bytecode at step **1005**. The current bytecode is specified by the current bytecode pointer.

At step **1007**, the system determines if the snippet has enough room to store the snippet code for the current bytecode and some continuation code. The continuation code is the native machine instructions that implement the equivalent of the advance and fetch steps in the interpretation cycle. If the snippet chunk has enough room, the system determines if a stop snippet flag is set in the bytecode table at step **1009**. The bytecode table is a table maintained by the system to store information about the various bytecodes. This table is shown in FIG. **13** and will be described in more detail later but for the purposes of this flowchart the bytecode table includes a flag which is set in the table for each bytecode to indicate to the system that upon encountering the bytecode, snippet generation should terminate.

At step **1011**, the system emits snippet code (e.g., native machine instructions) specific for the current bytecode. The bytecode specific snippet code may also be stored in the bytecode table as shown in FIG. **13**. The system advances the current bytecode pointer at step **1013**, and then returns to step **1005** to get the next bytecode to analyze.

If snippet generation is to be terminated, the system emits native machine instructions to increment the bytecode pointer at step **1015**. The bytecode pointer should be incremented by the number of byte used by the bytecodes which were placed in the snippet. The system then emits the continuation code at step **1017**. The continuation code is native machine instructions that jump to the address of the

interpreter that interprets the next bytecode. The continuation code may be the same for some bytecodes.

FIG. **13** shows a bytecode table that may be utilized to store information regarding different Java bytecodes. A bytecode table **1051** includes information regarding each of the bytecodes of the virtual machine instructions. In a preferred embodiment, the bytecode table is generated once when the Java virtual machine is initialized. As shown, a bytecode value **1053** (shown in decimal), name of the bytecode **1055** and size of the bytecode **1057** (number of bytes it occupies) are stored in the bytecode table. Additionally, a stop snippet flag **1059** as described in reference to FIG. **12** indicates whether the bytecode should terminate snippet generation when it is encountered.

The bytecode table may include a pointer to snippet code **1061** for each bytecode to which native machine instructions will be generated. Thus, as shown, a template table **1063** may be utilized to store templates for the native machine instructions for each bytecode. The template table allows for fast generation of snippets as the native machine instructions for the bytecodes may be easily determined upon reference to template table **1063**. Additionally, the templates of native machine instructions may also be used to interpret the bytecodes. Another column in bytecode table **1051** may indicate a snippet code size **1065** of the template in the template table.

CONCLUSION

While the above is a complete description of preferred embodiments of the invention, there is alternatives, modifications, and equivalents may be used. It should be evident that the invention is equally applicable by making appropriate modifications to the embodiments described above. For example, the embodiments described have been in reference to increasing the performance of the Java virtual machine interpreting bytecodes, but the principles of the present invention may be readily applied to other systems and languages. Therefore, the above description should not be taken as limiting the scope of the invention which is defined by the metes and bounds of the appended claims along with their full scope of equivalents.

What is claimed is:

1. In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:

receiving a first virtual machine instruction;

generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction; and

executing said new virtual machine instruction instead of said first virtual machine instruction.

2. The method of claim **1**, further comprising overwriting a selected virtual machine instruction with a new virtual machine instruction, the new virtual machine instruction specifying execution of the at least one native machine instruction.

3. The method of claim **2**, wherein the new virtual instruction includes a pointer to the at least one native machine instruction.

4. The method of claim **2**, further comprising storing the selected virtual machine instruction before it is overwritten.

5. The method of claim **2**, further comprising storing a back pointer to a location of the new virtual machine instruction.

6. The method of claim **2**, wherein the new virtual machine instruction includes a pointer to a data block in which is stored the at least one native machine instruction, a copy of the selected virtual machine instruction, and a back pointer to location of the new virtual machine instruction.

7. The method of claim **6**, further comprising generating the virtual machine instruction that were input by storing the copy of the selected virtual machine instruction stored in the data block at the location specified by the back pointer.

8. In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:

inputting virtual machine instructions for a function;

compiling a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instruction;

representing said at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the fuction.

9. A stored data structure of hybrid virtual and native machine instructions, comprising:

a sequence of virtual machine instructions for a function including a new virtual machine instruction;

at least one native machine instruction specified by the new virtual machine instruction for execution with the sequence of virtual machine instructions; and

a computer readable medium that stores the sequence of virtual machine instructions and the at least one native machine instruction;

wherein

the at least one native machine instruction is stored in a data block, and

the data block stores a copy of a selected virtual machine instruction that was overwritten in the sequence of virtual machine instructions by the new virtual machine instruction.

10. The stored data structure of claim **9**, wherein the new virtual machine instruction includes a pointer to the at least one native machine instruction.

11. The stored data structure of claim **9**, wherein the block stores a pointer to a location of the new virtual machine instruction in the sequence of virtual machine instructions.

12. The stored data structure of claim **9**, wherein the data block is stored in an array of blocks.

13. The stored data structure of claim **9**, wherein the at least one native machine instruction is generated from a compilation of a portion of the sequence of virtual machine instructions beginning with the selected virtual machine instruction.

14. The stored data structure of claim **9**, wherein the virtual machine instruction are Java virtual machine instructions.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,910,205 B2 Page 1 of 1
DATED : June 21, 2005
INVENTOR(S) : Bak et al.

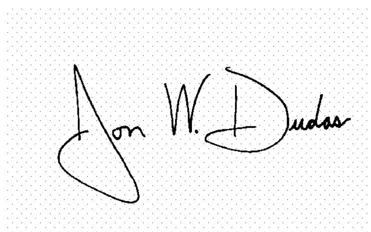
It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 13,
Line 59, change "new virtual" to -- new virtual machine --.

Column 14,
Lines 13 and 58, change "machine instruction" to -- machine instructions --.
Line 25, change "the fuction" to -- the function --.

Signed and Sealed this

Sixth Day of September, 2005



JON W. DUDAS
Director of the United States Patent and Trademark Office

Claims Involved in the Appeal of Reexamination Control No. 95/001,548

Claim 1: In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:

receiving a first virtual machine instruction;

generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction; and

executing said new virtual machine instruction instead of said first virtual machine instruction.

Claim 2: The method of claim 1, further comprising overwriting a selected virtual machine instruction with a new virtual machine instruction, the new virtual machine instruction specifying execution of the at least one native machine instruction.

Claim 3: The method of claim 2, wherein the new virtual machine instruction includes a pointer to the at least one native machine instruction.

Claim 4: The method of claim 2, further comprising storing the selected virtual machine instruction before it is overwritten.

Claim 8: In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:

inputting virtual machine instructions for a function;

compiling, at runtime, a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instruction;

representing said at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the function.

Claim 15: In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:

receiving a first virtual machine instruction;

generating, at runtime, a new virtual machine instruction that represents or references one or more native machine instructions that can be executed instead of the first virtual machine instruction;

overwriting, at runtime, the first virtual machine instruction with the new virtual machine instruction; and

executing the new virtual machine instruction and the one or more native machine instructions instead of the first virtual machine instruction.

Claim 16: In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:

receiving virtual machine instructions;

generating, at runtime, a new virtual machine instruction that represents or references one or more native machine instructions that can be executed instead of one or more of the virtual machine instructions;

overwriting, at runtime, a select one of the virtual machine instructions with the new virtual machine instruction, the new virtual machine instruction specifying execution of the one or more native machine instructions; and

executing the new virtual machine instruction and the one or more native machine instructions instead of the select one of the virtual machine instructions, the one or more native machine instructions performing a same operation as if the select one of the virtual machine instructions was executed.

Claim 18: In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:

inputting virtual machine instructions for a function;

compiling a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instructions;

representing the at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the function; and

overwriting, at runtime, a select one of the virtual machine instructions with the new virtual machine instruction.

Claim 19: The method of claim 18, further comprising executing the new virtual machine instruction, the executing of the new virtual machine instruction causing the at least one native machine instruction to execute instead of the select one of the virtual machine instructions.

Claim 20: The method of claim 18, wherein the at least one native machine instruction performs a same operation as if the select one of the virtual machine instructions was executed.

Claim 21: In a computer system, a method for increasing the execution speed of virtual machine instructions, the method comprising:

inputting virtual machine instructions for a function;

compiling a portion of the function into at least one native machine instruction so that the function includes both virtual and native machine instructions;

representing the at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of the function;

overwriting, at runtime, a select one of the virtual machine instructions with the new virtual machine instruction, the new virtual machine instruction specifying the at least one native machine instruction; and

executing the new virtual machine instruction and the at least one native machine instruction instead of the select one of the virtual machine instructions.

Case: 14-1351 Document: 1 Page: 110 Filed: 05/13/2014

CERTIFICATE OF SERVICE

I hereby certify that I electronically filed the foregoing with the Clerk of the Court for the United States Court of Appeals for the Federal Circuit by using the appellate CM/ECF system on May 13, 2014.

I certify that all participants in the case are registered CM/ECF users and that service will be accomplished by the appellate CM/ECF system.

Dated: May 13, 2014

/s/ Mehran Arjomand

CERTIFICATE OF COMPLIANCE WITH RULE 32(a)

This brief complies with the type-volume limitation of Rule 32(a) of the Federal Rules of Appellate Procedure because it contains 12,378 words.

Dated: May 13, 2014

/s/ Mehran Arjomand